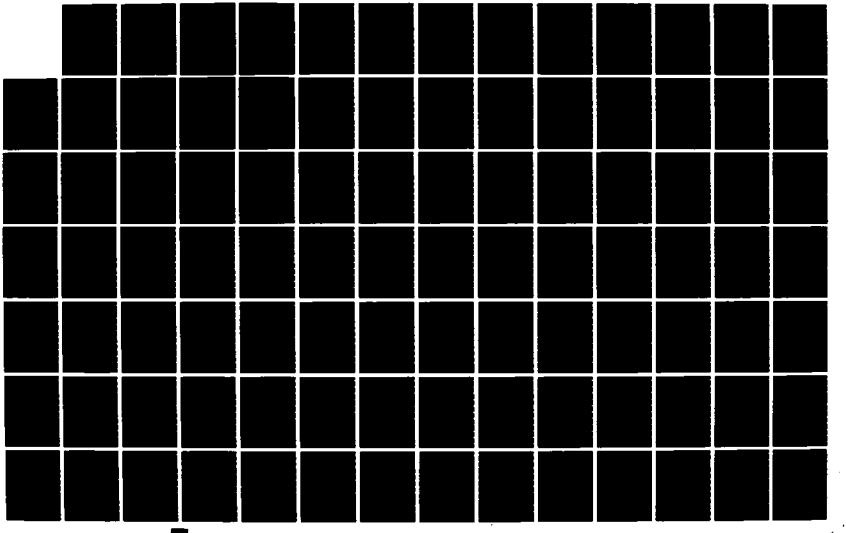
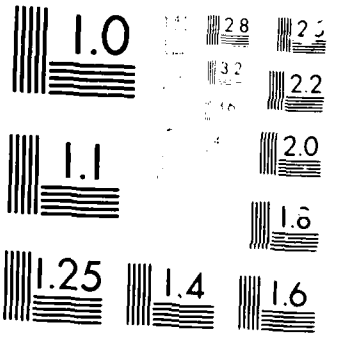
AD-A169 247 AVAILABILITY AND CONSISTENCY OF GLOBAL INFORMATION IN COMPUTER NETWORKS(U) CALIFORNIA UNIV BERKELEY C V [redacted]RTHY MAY 86 ARO-19159.3-EL DAAG29-83-K-0086 1/3

UNCLASSIFIED F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
2.0
1.8
2.2

ARO 19159.3-EL

(2)

AD-A169 247

Final Report
for Contract

DAAG29-83-K-0086

DTIC FILE COPY

REPORT DOCUMENTATION PAGE

... in the face of various kinds of
... information in the presence of

... , in which failures are
... , in which an indefinite period
... to fail.

... clock facility was

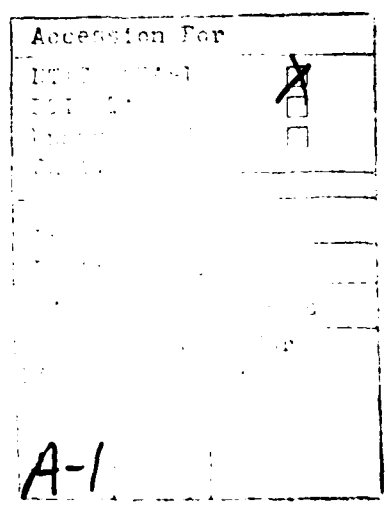
... the correctness of
... was developed.

... rithms were developed

AVAILABILITY AND CONSISTENCY OF GLOBAL INFORMATION IN COMPUTER NETWORKS

FINAL REPORT

C. V. Ramamoorthy



AVAILABILITY AND CONSISTENCY OF GLOBAL INFORMATION IN COMPUTER NETWORKS

ABSTRACT

A principal feature of computer networks is the ability of the various sites of the network to access and update shared information. At the application level, the global information takes the form of shared file systems, databases, etc., and at lower levels, it takes the form of status information used in controlling the network. This report focuses on techniques for maintaining (i) the availability of global information in the face of various kinds of failures and (ii) the consistency of global information in the presence of concurrency.

Two failure models are considered: the crash model, in which failures are instantly detected, and the malfunction model, in which an indefinite period of time may lapse before failures are detected. In the latter model, failed sites in the network may execute arbitrary state transformations and emit arbitrary messages; they may exhibit malicious intelligence trying to disrupt the functioning of the rest of the network.

A network status maintenance scheme based on a global clock facility is designed for the crash model. The global clock is a system of local clocks, one at each site. Using the primitives provided by this scheme, desired consistency and availability attributes can be provided for higher-level software. A feature of this scheme is the guarantee that if site A has site B marked DOWN at a certain time, then site B is really DOWN at that time. An algorithm for updating a replicated file is constructed using the scheme.

For the malfunction model, an approach to maintaining the correctness of global information and preventing error propagation is developed. A combination of an assertion-checking step and a unanimity-reaching step is used, along with replication of the global information at multiple sites, for this purpose. The requirements of the unanimity-reaching step are formalized as a more general form of the Byzantine Generals Agreement and algorithms for reaching this generalized agreement are presented.

Centralized and distributed deadlock detection algorithms are developed for distributed databases. These algorithms use clock facilities to ensure the consistency of 'snapshots' taken of the status of resources and transactions. It is shown that all genuine deadlocks are detected and no spurious indications of deadlock are given by these algorithms.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

1.1. Definition of Global Information

A principal feature of computer networks, both an advantage and a necessity, is the ability of the various sites in the system to access and update shared information. We refer to such shared information as *global*. At the application level, the global information takes the form of shared file systems, databases, etc. At lower levels, it takes the form of *control* or *status* information for the purpose of such system functions as resource management, synchronization, routing, reconfiguration, protection and error control. In this thesis, we will focus on certain requirements arising in the management of global information and developing techniques for satisfying them. Our attention will be restricted to distributed computer systems that use message-passing rather than shared-memory as the basic communication mechanism.

Just as with information stored in non-distributed systems, it is necessary to manage global information in such a way that certain desirable attributes are ensured. The major attributes are:

(a) rapid accessibility: it should be possible to access and update the global information with low latency and high bandwidth.

(b) security: no unauthorized entity should be able to access or update the information.

(c) integrity: relevant invariants, e.g., functional dependencies between different items of information, restriction to a set of permissible values, etc.,

should be preserved.

(d) availability: in spite of failures of parts of the network, it should still be possible to access and update the information.

(e) consistency: in spite of concurrent usage of the information, every entity that accesses the information should be able to form a consistent view of it.

Our research is concerned with the closely-interacting attributes of availability and consistency. Our objective is to develop techniques for

(i) ensuring the availability of global information in the face of various kinds of failures, and

(ii) ensuring that the entities that access the information get a consistent view of it, in spite of concurrency.

1.2. Availability

The availability of a system is usually defined as the probability that the system will be functioning normally at any time during its scheduled working period [BAR 65]. We can extend this definition and say that the availability of a piece of global information for a given operation (e.g., read, update) is the probability that an entity authorized to perform that operation is able to do so. In order to make this definition meaningful, we must include the proviso that the operation is carried out in a manner that satisfies certain requirements, e.g., the information obtained in a read operation should not be out-of-date or corrupted as a result of prior failures.

The difficulty of ensuring the availability of information depends on the kind and degree of failure that must be prevented from making the information unavailable. We distinguish two models of site failures – the *crash* model and the *malfunction* model. (Lamport makes the same distinction in [LAM 78b] using different terminology.)

In the crash model, if a site fails at time t,

(i) the site stops executing at t; any message it was in the process of sending is not received, or if received, is recognized as mutilated.

(ii) the internal state and contents of the site's non-stable storage are lost; the contents of the stable storage remain intact.

(iii) while in the crashed state, the site does not execute any operations.

(iv) on recovery, the site knows that it has been in the crashed state and initiates the appropriate recovery procedures.

(v) the site executes correctly between crashes.

The *fail-stop processors* of [SCH 83] exhibit essentially the above characteristics in their failure behavior. Consider a set of sites $\{R\}$, called *receivers*, which wish to obtain a piece of information from another site T, called the *transmitter*. If T *crashes*, it can result in a subset of $\{R\}$ receiving the information and the remaining sites in $\{R\}$ not receiving it. However, a crash in T cannot result in T sending out incorrect information to any receiver in $\{R\}$; crashes are benign failures in this sense.

In the malfunction model, failures are more general and dangerous, since failed sites can execute arbitrary state transformations and send arbitrary messages. If T *malfunctions*, it may give out incorrect information to sites in $\{R\}$ and it may also give out different values to different sites for the same piece of information. The protocols required to ensure that the receivers reach some form of agreement on the received value are more complex in this case. A site may malfunction because it has been taken over by a malicious agent or because of some undetected hardware or software error.

In order to show how the protocols for dealing with malfunctions must differ from those for crashes, we consider the example of a transactional distributed database. Assume that the crash model holds. From the viewpoint of preserving the availability of information in the database, the following protocols used in managing it are relevant:

(a) *read and update protocols*

A number of protocols exist for performing the read and update operations. The selection of the protocols to be used in a design will affect the availability of the information for read and update operations. For example, the read protocol may involve reading any single copy of the information and the update protocol may involve updating all copies. Alternatively, we could read x copies and update y copies where $(x+y)$ is greater than the total number of copies, using a timestamping mechanism to determine the most up-to-date value if some of the x values read were not coincident. Yet another alternative would be to direct all reads and updates to a single copy, called the *primary*, which is responsible for distributing updates to all the other copies. These protocols ensure different degrees of availability for read and update operations, as well as other attributes such as response time, communication costs, etc.

(b) *commit protocols*

Commit protocols (such as the 2-phase commit protocol [GRA 78]) are used to ensure the atomicity of transactions, i.e., either all the effects of the transaction are installed (*commit*) into the global information structures concerned or none are (*abort*). From the viewpoint of preserving availability, the important question is whether or not the protocol is *nonblocking* [SKE 81]. A commit protocol is said to be non-blocking for a class of failures

the Byzantine Generals Agreement

has been shown to be inherently

appropriate manner and degree of its

determined. The most comprehen-

to date is [LAM 81a] in which com-

and data at every site is assumed. In

partially replicated, case and propose

of malfunctions against the costs

requirement for global informa-

a number of concurrently executing

that operate in various ways on the global infor-

operates on the global information through atomic

effect of the execution of each atomic action on the information

effectively has exclusive access to it during

For example in a database context, the transaction is made

actions. Another example is the allocation table for a

resources. The transactions that share the resources perform

and release actions on the resources thus altering the allocation

the changed status of the resources. If deadlock-detecting

exist at one or more sites, they will need to execute read actions

table. An atomic action "sees" the effect of other atomic

that have executed earlier, e.g. in the database example, a read

effect of prior update actions

Consistency requirements are restrictions placed on the permissible interleavings of actions of the concurrent transactions and thereby on what an atomic action may be allowed to see. In the database example, one requirement that is usually made is that a read action of a transaction see the effects of all actions of another transaction or see the effects of none of them. In the example of the resource allocation table, a requirement may be that if a read action of the deadlock detector sees the effect of an acquire action by a transaction, it should also see the effects of any resource release executed by the transaction prior to the acquire operation. Otherwise, an inconsistent picture of the status of resources and transactions may be formed by the deadlock detector, resulting in the detection of false deadlocks.

The techniques that can be used for ensuring consistency in distributed databases have been elegantly classified in [BER 81]; they fall into the two broad categories of *locking* and *time-stamp ordering*. The answer to the question of which technique is preferable for distributed databases with given requirements must await further research. [CAR 83] suggests, on the basis of investigation of a restricted set of locking and timestamp-ordering algorithms, that the former may be superior for single-site databases. On the other hand, [GAL 82] and [LIN 83] find that timestamp-ordering is superior for distributed databases, at least in some environments. Whatever the answer may be for databases, it can be concluded from the available literature that timestamp-ordering appears to provide a more versatile and efficient mechanism for preserving the consistency of global information used in the control of the distributed system. Examples of areas in which timestamp-ordering has been used are database synchronization [BER 81], network status maintenance [HAM 80], deadlock detection [TSA 82], dynamic

reconfiguration [MA 81], etc. In Chapters 2 and 4, we investigate the use of clock facilities in network status maintenance and deadlock detection respectively.

Some sort of clock mechanism has to be available from which the time-stamps can be derived. Such a facility should have the following characteristics:

(i) it should be distributed for reasons of reliability, survivability and efficiency of access.

(ii) the facility should assign time values which reflect the ordering of events in the computer system. For most applications, it would be sufficient if the values reflected the ordering of events at a single site and the ordering of events at different sites imposed by the flow of messages.

(iii) the value of the clock at a site should be close to the real-world time. This in turn implies that clock values at different sites should not drift appreciably from one another.

(iv) Maintenance of the clock facility should be inexpensive.

[LAM 78a] has proposed distributed clock mechanisms synchronized by messages which satisfy properties (i) and (ii). The mechanisms require the clock at a site to be advanced when a message arrives bearing a timestamp value greater than the local clock value. Hence, all the clocks have a tendency to catch up with the fastest one among them. This in turn may cause them to drift ahead of the real-world time. However, turning them back may vitiate the required ordering of events. [BEL 79] suggests a *slowing down* of clocks, when too large a drift from the real-world time is noticed. This would provide property (iii). Much of the functionality required of the facility could

be implemented by hardware or microcode and this could help to satisfy the efficiency requirement mentioned in (iv) above. [HAM 80] proposes techniques for including site crashes among the events that the clock mechanism imposes an order on. It relies upon the use of probe messages by means of which a site in the network can ascertain the health of any other site in the network. The defects of this approach and an alternative solution are covered in Chapter 2. The problem of synchronizing clocks in the malfunction model is covered in [LAM 81b].

1.4. Providing Facilities for Availability and Consistency

There is a paucity of systems that have implemented any but the simplest options for providing availability and consistency. SDD-1 [HAM 80] is one example of experimentation with a novel distributed operating system (the *RelNet*), which has attempted to provide timestamp based lower-level mechanisms on the basis of which the availability and consistency requirements can be fulfilled. But more experience with similar systems, which select from the various options for providing availability and consistency to achieve viable combinations, is required.

1.5. Scope of Report

As mentioned above, it is our belief that timestamp-ordering based on a global clock mechanism provides the most versatile basis that can satisfy the consistency requirements of both application and network control functions. In the absence of failures, such a clock mechanism is simple to build [LAM 78a]. However, much of the difficulty in controlling distributed systems lies in the problem of providing failure-tolerance, which is closely intertwined with the consistency problem. As will be seen in Chapter 2, it is more

difficult to construct a clock mechanism that assigns times to failure-related events in such a way as to make it useful for satisfying consistency *and* failure-tolerance requirements. In Chapter 2, we provide a design for such a clock mechanism for the crash model. The design gives each site a view of the status of all the sites in the network at every instant of time. The network status view is used to construct a solution to the problem of updating a replicated file. This solution involves keeping some of the replicas continuously up-to-date, while the others are updated periodically. When sites holding the up-to-date replicas crash, they have to be replaced. The synchronization requirements of this solution provide a good test of the capabilities of the clock mechanism.

In Chapter 3, we address the problem of preventing error propagation in global information due to malfunctions. A more general form of the Byzantine Generals Agreement is formulated and methods for adapting it to provide different degrees of malfunction-tolerance, according to the criticality of the global information, are developed.

Chapter 4 deals with deadlock detection in distributed database systems. The race conditions that render most of the algorithms in the literature incorrect are discussed, and algorithms making use of a clock facility are proposed. This feature of the algorithms helps in showing that all genuine deadlocks are detected and no spurious indications of deadlock are given.

CHAPTER 2

DESIGN AND USE OF A NETWORK STATUS MAINTENANCE SCHEME

2.1. Introduction

In this chapter we propose a technique for maintaining information about the operational status of the sites in a point-to-point network, e.g., the Arpanet. We show how this technique can provide the basis for the solution of a control problem concerned with file updating in such a network.

In the network, as different sites go out of operation and recover, the allocation of functions and tasks must be changed in accordance with network status in order to preserve the services the network provides. For this purpose, a view of the status of the various sites must be obtained and updated as time proceeds.

In order to co-ordinate these functions and tasks as well as to ensure the consistency of the view of system status, a synchronization mechanism is necessary. The mechanism used in our method is a *global clock facility*.

In Section 2.2, we discuss the issues concerned in status maintenance and the proposed method. In Section 2.3, we show how a reconfiguration control problem in file updating can be solved using this method.

2.2. Network Status Maintenance

2.2.1. Overview

Section 2.2.2 discusses the requirements placed on the global clock facility in order that it may serve as a synchronization mechanism for our

status maintenance scheme, along with previous work in designing such a facility. Section 2.2.3 describes previous work in status maintenance. Section 2.2.4 develops the proposed method.

2.2.2. Requirements for the Global Clock Facility

In constructing a global clock facility, we must ensure that it is consistent with the notion of *causality*. If an event X causally affects another event Y, the global clock should assign a greater time to Y than to X.

Consider a failure-free distributed system. An event can causally influence other events occurring after it at the same site e.g. a write operation on a piece of data will influence the result of a subsequent read operation. Again, when a message is sent from one site to another, an event occurring before the sending of the message at the first site can influence events occurring at the second site after the receipt of the message.

In order to achieve reliability and for efficient accessibility, it is desirable to construct a global clock out of several local clocks, one at each site. Events at a given site are assigned times using the current value of the local clock. In order to ensure that these assignments satisfy the causal relationships among events arising in the two ways mentioned above, Lamport [LAM 78a] proposed two rules which each local clock should obey:

> $C1$. At each site i, the local clock $C(i)$ is incremented between any two successive events.
>
> $C2$. If event a is the sending of a message m by site i, then the message contains a timestamp, the time assigned by $C(i)$ to a. Upon receiving the message, site j sets $C(j)$ to a value more than the maximum of its current value and the timestamp. The receipt of m is

supposed to occur after the setting of $C(j)$.

If these rules are followed, the causal relationships between events will be reflected by the times assigned to them.

Next consider the case where site failures are among the events to be considered. If a site X fails, the time at which another site Y detects the failure and marks X as *DOWN* should be greater than the time on X's clock when it failed. Similarly the time on Y's clock when it marks X *UP* on its recovery should be greater then the value on X's clock when it recovers; however there are some additional considerations relevant here which we discuss below.

Consider a distributed system of two sites, X and Y. Assume that both sites are operational and that Y wants to perform a read operation, to which it has assigned the time T, on a local file. Further assume that the result of the read operation at T should reflect the effect of all updates to the file assigned times prior to T. (Note that this requirement is not implied by causality: while all updates which do influence the read operation are required by causality to have earlier times, not all updates with times less than T are required to make their influence felt when the read operation is performed i.e. they may be performed after the read.) To satisfy these requirements, Y waits till it receives a message from X timestamped greater than T (if desired, it could send a message timestamped T with a request for acknowledgement). Assuming that messages are delivered from one site to another in the order sent, and that a site sends its messages in timestamp order, Y knows now that it has now received all update messages originating from X which have update times less than T. It can perform all such updates (local and from X) and then perform the read operation.

Now assume instead that X failed some time prior to T and is known to have done so by Y when it performs the read operation. If X recovers after the read operation and issues an update timed less than T, the results of the read operation will not fulfill the specified requirement. For this reason it is desirable that on recovering, X should set its clock to a value greater than any at which Y has it marked as *DOWN*. Earlier we saw that in order to satisfy causality, the time at which X recovers should be less than that at which it is marked *UP* at Y. Now we see that X should recover with a clock setting greater than Y's clock value when it marked X *UP*. These two requirements can be reconciled by assuming that X pauses, i.e. does only null operations till its clock value exceeds that at which Y marked it *UP* (Fig. 2.1).

Summarizing, we see that our global clock facility should obey rules $C1$ and $C2$ and a third rule:

> $C3$. If a site i is marked *DOWN* at time t at another site j then site i should not be operational at that time t.

2.2.3. Previous Work in Status Maintenance

Kuhl and Reddy [KUH 80] propose a scheme in which a site is tested by a subset of its immediate neighbors who pass the test results to the rest of the network. Only the test results sent by those sites which themselves have been found to be operating correctly are relayed through the network. The deficiencies of this scheme are:

(i) there is no notion of time attached to the test results so that it is difficult to integrate the test results from different sites in a consistent manner and to determine for what period they are valid.

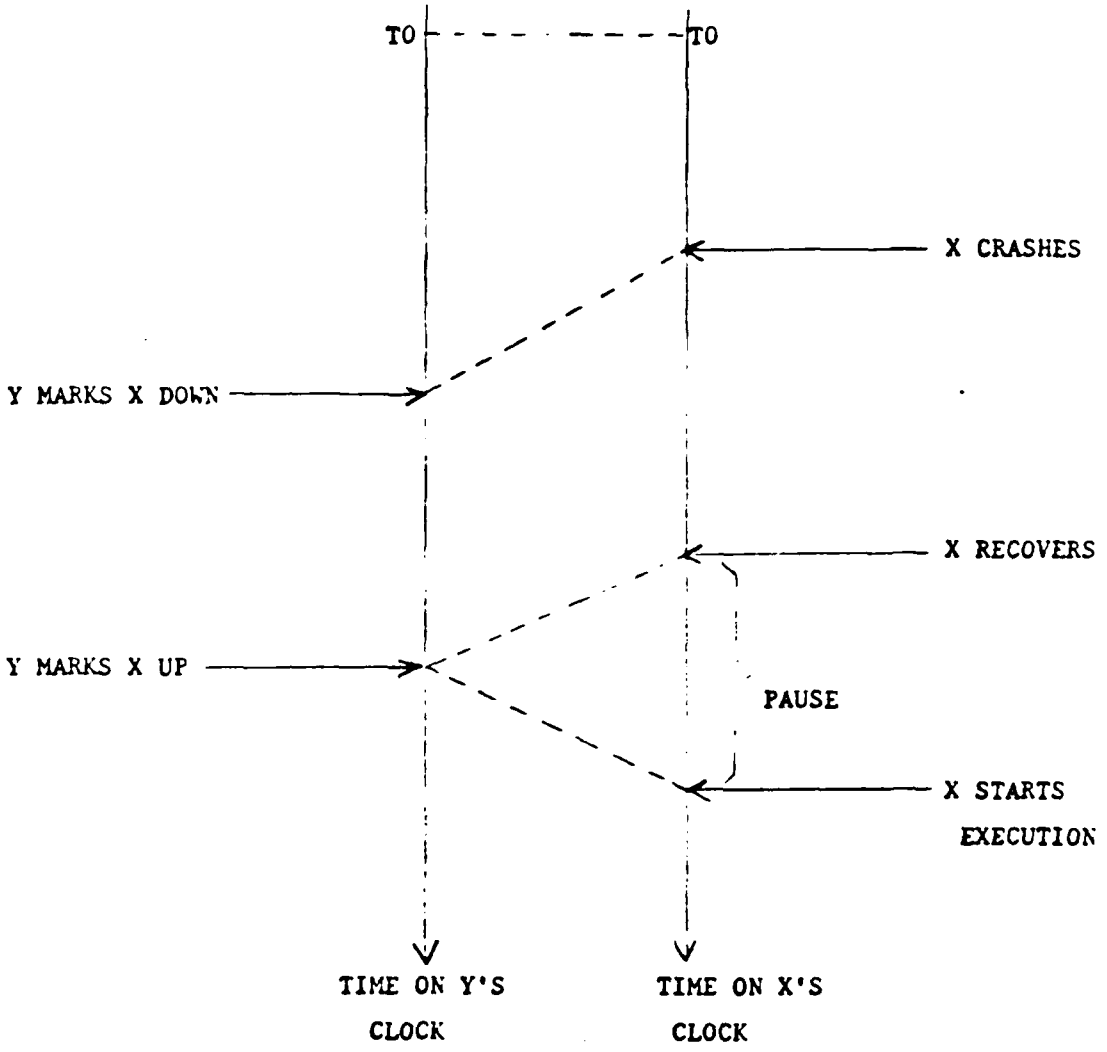


FIG. 2.1. EXAMPLE TO ILLUSTRATE RULE C3

(ii) the question of link failures, which may cause correctly operating sites to arrive at different conclusions concerning the status of a common neighbor, is not considered.

The SDD-1 RelNet [HAM 80] performs site status maintenance using a global clock facility which achieves the requirements described in the previous section. In this scheme, any site X in the network directly determines the status of any other site Y in the network by trying to communicate with it. If no response is obtained within a certain time, X marks Y as *DOWN* in its local status table. A *YOU_ARE_DOWN* message is sent to Y in case the lack of response were due to some other cause than a failure of Y. Receipt of this message causes Y to cease operation in order to comply with rule $C3$, and then to execute a recovery procedure. To ensure that Y actually gets the *YOU_ARE_DOWN* message it is deposited with another site called a *guardian* of Y, with whom Y periodically checks for such messages.

The defect of this status maintenance scheme is that sites may often be made to cease operation needlessly. If the network becomes congested at some spots, timers will begin to run out and sites will become busy, stopping operation themselves and recovering, thus aggravating the problem. A site may be too busy to reply in time to all the messages that it may receive from various parts of the network, but it may well be able to sustain a low-level protocol with its immediate neighbors to assure them that it has not failed. Other reasons for a site not responding in time could include being in a critical section, in a recovery procedure, in a high-priority task, etc. To force the site to cease operation in such situations is evidently not desirable. In the RelNet, it is possible that two or more sites trying to recover at the same time will force each other to stop operation repeatedly unless such a situa-

tion is detected and a random wait period observed before trying to recover again. The scheme of Kuhl and Reddy has distinct advantages in that the failures detected and communicated are much less likely to be spurious.

Our method overcomes the deficiencies of both the above schemes. In addition, it has a limited ability to deal with network partitions, which neither of the above schemes has.

2.2.4. Proposed Scheme

2.2.4.1. Overview

In the proposed scheme, every site periodically broadcasts the state of each communication link attached to it to the whole network. The state of the communication link may be broadcast as down either because the link itself has failed or because the site at its other end has failed. The state of a given site is determined by other sites in the network on the basis of the states of all the links attached to that site. This requires putting together reports from different sites, in a consistent manner. This is done with the help of a global clock facility which fulfills the requirements stated in Section 2.2.2.

Consider a network N composed of two parts N_1 and N_2 connected by the set of links L (Fig. 2.2). Suppose the sites in the part N_2 which are connected to the links L observe that the links have failed. This information is circulated among the sites in N_2. Suppose further, that the sites in N_1 form a minority (usually just one site). Then the sites in N_2 will mark the site(s) in N_1 as *DOWN* on the basis of this information.

It may be that the sites in N_1 have actually failed, causing the links L to appear to have failed to the sites in N_2. On the other hand, it may be the

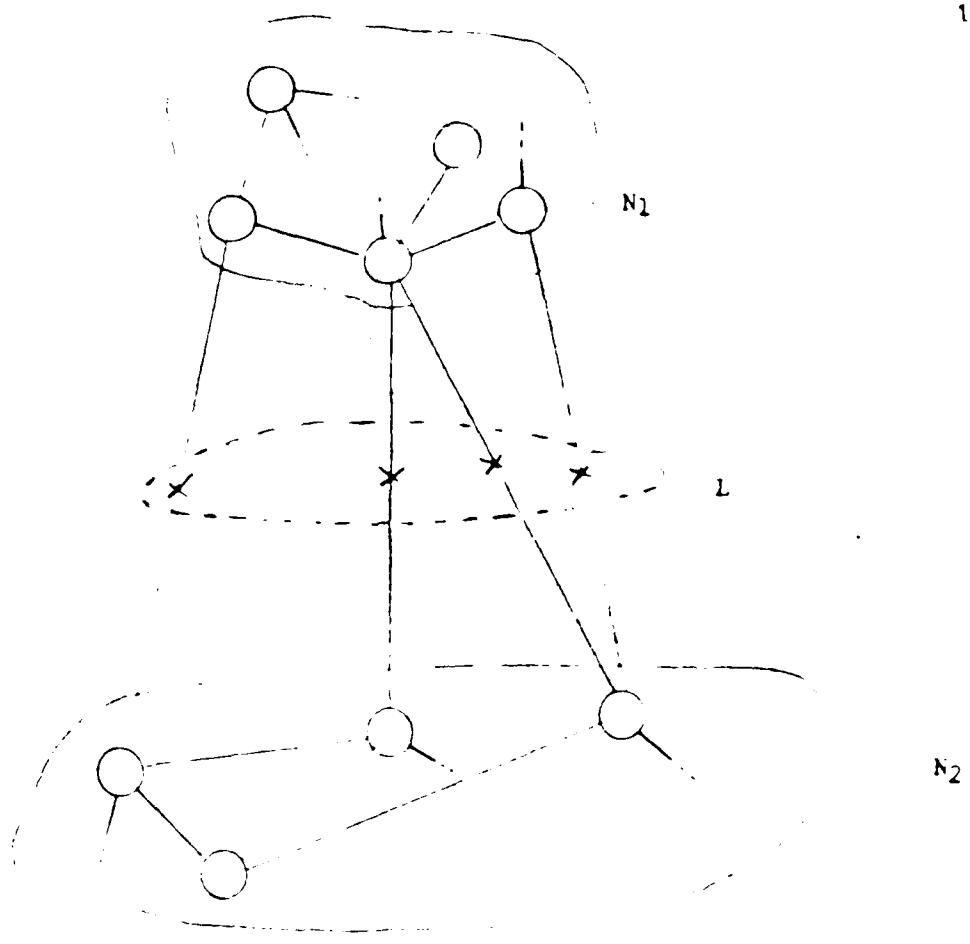


FIG.2.2. SCENARIO FOR SITES IN N_1 BEING MARKED DOWN BY SITES IN N_2.

links that have failed. Further, not all the links may have failed simultaneously, but the news of the recovery of some of them may not have reached all the sites in N_2 when they made the decision to mark the sites in N_1 *DOWN*. In the latter two cases, the sites in N_1 connected to the links L detect the failures of the links and circulate the information. By this means, the sites in N_1 realize the possibility of their being marked *DOWN* and hence cease operation in time to comply with rule *C3*.

The sites in N_1 execute their recovery procedure as follows. First the sites that are directly connected with operational sites in N_2 complete their recovery and enter normal operation. Then their neighbors who had no direct connection with an operational site till then, are able to start and complete their recovery and enter normal operation. This process continues till all sites in N_1 recover.

The recovery of any given site i is performed by first informing the network that its links are functioning and that it itself is about to resume normal operation. In broadcasting this information, site i should not have to wait for failed sites to recover and acknowledge that they have received this information. One possible way out of this difficulty would be to look at the circulating information concerning link failures in the network and use it to mark sites *DOWN* as above. Then site i need only wait for acknowledgement messages from sites not marked *DOWN* stating that they know about its impending transition to normal operation. The sites thus marked *DOWN* are assumed to mark every site *UP* when they initialize themselves on recovery. However, this method is a double-edged sword for site i. Other broadcast messages of site and link recoveries occurring at the same time may not reach site i since the broadcasters of these messages, who may have site i

marked as *DOWN*, would not have to ensure that the broadcast messages reach site i. Hence site i may form an incorrect picture of which sites are *DOWN* and thus omit to inform sites that are in normal operation of its transition to normal operation, leading to a violation of rule *C*3. Our solution to this problem is for each site in its recovery procedure to have one or more sites in normal operation to serve as *guards* in ensuring that broadcasts reach site i. This is in contrast to the RelNet technique. There, if a site i concludes that another site j is *DOWN*, site i has a *YOU_ARE_DOWN* message sent to site j. This ensures that the latter ceases operation in time to validate site i's mistaken assumption, if it has not really failed.

2.2.4.2. Assumptions

The following assumptions are made:

(i) The network has a fixed topology with links connecting pairs of sites as in the Arpanet and each site knows this topology. This assumption, as well as the use of a network-wide broadcast facility in our scheme, limits the size of the network to which it can be efficiently applied. For large networks, a hierarchical scheme will have to be used.

(ii) If partitions occur, they occur in such a manner as to leave a majority of sites connected. This assumption is required because our method handles partitions as follows.

When the network gets partitioned, the partition(s) that have a minority of sites cease operation. This is done because the sites in the majority partition (if one exists) will mark the sites in the minority partitions to be *DOWN*. Hence, in order to comply with rule *C*3, the sites in the minority partitions must cease operation until the partition is repaired, while the sites in

the majority partition continue in operation.

Thus if the network partitions into more than two pieces with each piece only having a minority of sites, all sites cease operation. Even when the partition is repaired it is difficult for the network to bring itself up automatically after this event for the following reason. If a majority of sites is always operational, they enable a failed site (or group of sites), on recovery to make the necessary deductions about the clock values with which sites which are still in failed states, went down. The recovering sites are then able to set their local clocks to values which ensure compliance with rule *C*3 and then resume normal operation. But if all sites cease operation at some time, it is difficult for any of them, when the partitions are repaired, to recover and set their clocks to such values until all sites have recovered and their clock values have been ascertained. Our method does not handle the problem of a network in which all sites have ceased operation, and will have to be extended to deal with such a situation.

(iii) For similar reasons, we assume that the number of sites that have failed is small enough to leave at least a majority of sites which are connected, in operation. Otherwise the same catastrophe, namely, of all the sites ceasing operation, will occur.

(iv) Sites are assumed to *stop* when they fail i.e. they do not fail in such a way as to execute their algorithms incorrectly or exhibit malicious behavior. Thus it is the *crash* model of Chapter 1 that we are assuming.

2.2.4.3. Site and Link States

A site or link is simply marked as *UP* or *DOWN* by every site in the network in the data structures that it maintains to record its view of the system state. However, in addition, a site itself maintains more detailed state

information concerning itself as it goes through the various stages of recovery and normal operation. Similarly, the sites attached to a link also maintain more detailed state information concerning the link. In this section we summarize this detailed state information. The data structures that are used to record system state views are discussed in Section 2.2.4.8.

Fig. 2.3 shows the states and state transitions that a site may go through as recorded in the site itself. The site enters the *crashed* state when it actually crashes because of a hardware or software fault or when it suspects that some other site may consider it *DOWN* (i.e. it crashes itself). *Sync* and *pause* are recovery states. In the *sync* state, the site synchronizes its logical clock with the clocks of neighboring sites which are in the normal *operational* state. In the *pause* state, the site informs other sites in the network that its links are functioning properly and that it is about to enter the *operational* state. Only when the site enters *operational* state does the higher-level software (e.g. the file-updating software described in Section 2.3) resume execution. The site may reenter the *crashed* state at any instant for either of the two reasons given above.

Fig. 2.4 shows the states and state transitions for links. Consider a link connecting two adjacent sites i and j. Although this is in reality one bidirectional link, the two sites maintain their view of the state of this link in the form of the state of the unidirectional links (i,j) and (j,i) respectively. In the sequel, we will refer to the actual bidirectional link as a *bilink* whereas the unidirectional links whose states are recorded in the detailed state information alluded to above and in the data structures described in Section 2.2.4.8 are referred to as *unilinks*. When the bilink itself fails (e.g. due to hardware problems or noise) or when one of the sites connected to it

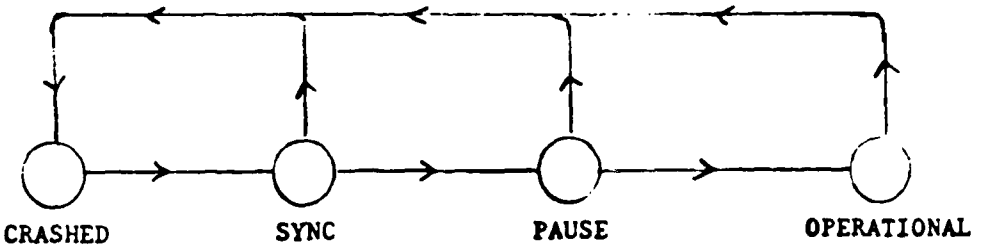


FIG. 2.3. STATE TRANSITIONS OF A SITE

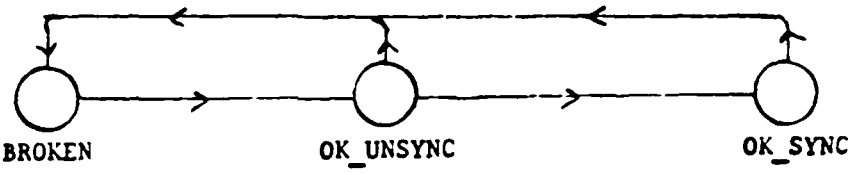


FIG. 2.4. STATE TRANSITIONS OF A UNILINK

crashes, the state of the corresponding unilinks is set to *broken* at both or one of the sites depending on which of the above situations exists. *Ok_unsync* is a recovery state, in which the clocks at the two ends of the unilink have not been synchronized, but the link is physically in usable condition. *Ok_sync* is the normal operational state. The link may enter *broken* state at any moment for either of the reasons given above.

2.2.4.4. The Clock Synchrony Rule

Let $C(i)$ denote the local logical clock (there is also a local real-time clock to be discussed in Section 2.2.4.5.) at site i, and $N(i)$ denote the set of immediately neighboring sites of i. For each site k in $N(i)$ site i maintains a register $LTR(k,i)$ which contains the largest timestamp attached to a message received by site i from site k over the bilink between them. The following relation is always maintained:

$$CSR: \quad C(i) < \Delta + \min \{ LTR(k,i): k \text{ in } N(i) \text{ and } st(i,k) = ok\_sync \}$$

where $st(i,k)$ is the detailed state of unilink (i,k).

This implies that, if at any instant two adjacent sites i and j record the unilinks (i,j) and (j,i) respectively as in *ok_sync* state, then at that instant:

$$|C(i)-C(j)| < \Delta$$

The local clock $C(i)$ at a site i may have to be advanced for several reasons. It is incremented by 1 for generating a new timestamp, and when a message arrives with a timestamp greater than the current value of $C(i)$, $C(i)$ must be increased to a value beyond the timestamp, if not already greater than the timestamp. These advances of $C(i)$ are required to implement the rules $C1$ and $C2$ described in Section 2.2.2 to satisfy causality. A clock advance may be necessary also when other events occur e.g. the

timer which keeps $C(i)$ in rough synchrony with a real-time clock runs out, a recovering site needs to bump its clock ahead to synchronize with its neighbors, etc.

If the advance cannot be made in compliance with *CSR* with the current values of the *LTR* registers, the site i sends a timestamped *REQ_TIME_SIGNAL* message to the appropriate neighbors depending on the values in the *LTR* registers, the current clock value and the value to which it has to be bumped. The neighbors will reply each with acknowledgements bearing timestamps greater than the one attached to the *REQ_TIME_SIGNAL*. The acknowledgements will increase the values in the *LTR* registers, permitting $C(i)$ to be advanced. Note that $C(i)$ cannot be increased by more than Δ at a step so that greater increases have to be performed in multiple steps.

2.2.4.5. Synchronizing with Real-Time Clocks

It is desirable to keep each logical clock $C(i)$ in rough synchrony with a local real-time clock. This is necessary so that the local clocks do not drift apart to the degree allowed by *CSR* (two sites can be as far apart in their clock readings as the number of hops in the shortest path between them multiplied by Δ). Otherwise frequent *REQ_TIME_SIGNAL* messages will have to be sent in order to receive messages in accordance with rule *C2* and at the same time maintain *CSR*. The method of ensuring this rough synchrony depends on how well the real-time clocks at different sites are themselves synchronized with respect to each other. We consider two cases:

(i) Close Synchrony: Here the real-time clocks develop differences of the order of Δ only over very long periods of time. In this case, the SDD-1 Rel-

net method can be used [HAM 80]. Once after every interval of duration τ_c (say 1 second) on the real-time clock, its reading in seconds multiplied by a large number N (say 10^6), is compared with the logical clock and if the reading of the logical clock is less, it is set to the multiplied value. Otherwise no action is taken. Usually, however, the increment in the logical clock during the period τ_c is much less than $N^*\tau_c$ and the setting does occur. When the real-time clocks develop differences of the order of an appreciable fraction of Δ, they should be resynchronized in some manner.

(ii) Loose Synchrony: Here the real-time clocks may develop differences of the order of Δ comparatively quickly. In this case, every τ_c interval the reading of the logical clock $C(i)$ is stored away in a location $RC(i)$. When the next such interval elapses, the logical clock reading is compared with $RC(i)+N^*\tau_c$ and, if less, is replaced by the latter. The new value of $C(i)$ is stored in $RC(i)$. In this way, a steady increase of $C(i)$ with respect to real-time is obtained even if the real-time clocks are only in loose synchrony. Use of the previous technique could result in sudden disruptive jumps in $C(i)$ when messages from other sites arrive, if the real-time clocks develop large differences.

Increments in $C(i)$ arising from the synchronization with the real-time clock can be anticipated and *REQ_TIME_SIGNAL* messages sent out in advance as in the RelNet [HAM 80], so that incrementation does not get held up because $C(i)$ cannot be advanced in consonance with *CSR* using the current *LTR* values.

2.2.4.6. The Link Monitor Module

The basic function of this module is to probe each unilink periodically, and to warn other interested parties when the unilink appears to go dead and

when it recovers.

Each site executes this protocol module on each of its unilinks to immediate neighbors. If a neighbor does not respond in timely fashion to the messages of this protocol, it runs the risk of having the unilinks to it marked *DOWN* and then the site itself may be marked *DOWN*. This may cause the site to have to cease operation in order to comply with rule *C*3.

The messages of this protocol are not timestamped since it is required to execute when the clock has not been synchronized with the clocks of neighbors during recovery. Again, it may be that the site is waiting for an increment in its *LTR* registers in order to increase $C(i)$. The protocol is required to be sending messages during this waiting period, too. Therefore, the protocol must have some independent sequencing mechanism to correlate messages sent with their responses.

The protocol requires every site i to periodically test each unilink directed from site i by sending a *REQUEST* message to which an *ACK* response is expected from the receiving site at the other end of the unilink within some time-out period. If the *ACK* is not received before the timer runs out, site i sets the unilink to *broken* if it is not already in that state. The probing of the unilink is continued when the unilink is in *broken* state. When the site i next receives an *ACK* to a *REQUEST*, it sends out a *LINKDOWN* message to ensure that the neighbor realizes that the unilink from site i to it was in *broken* state. To this message a *LINKDOWN_ACK* response is expected. If it arrives in the time-out period, the unilink (i,j) is set to *ok_unsync* state. Symmetrically, if a *LINKDOWN* message arrives at site i, it sets the unilink (i,j) to *broken* state, sends a *LINKDOWN* message if it has not already sent one to which a response is pending, and then sends a

LINKDOWN_ACK response. The probing of the unilink goes on in the *ok_unsync* state, also.

If a message not related to this protocol arrives over a unilink (j,i) to site i while it has the unilink (i,j) marked as *broken* then the message is suppressed. Also site i itself is prevented from sending a message over the unilink (i,j) while it is in *broken* state.

The link monitoring module provides a facility by which an interrupt is generated to any process in site i which has requested to be informed when a unilink (i,j) goes into *broken* state or comes back to *ok_unsync* state. If the unilink is already in the state specified an immediate return is provided. Fig. 2.5 shows the unilink state transitions in which the link monitor module in involved.

2.2.4.7. The Link State Reporter Module

The function of this module at a given site is to watch the state of the unilinks directed from the site and to broadcast the state to the network.

Every τ_L ticks of the local logical clock, at a site i in the *operational* or the *pause* state, this module broadcasts the state of all the unilinks directed from it with a timestamped message.

In addition, when a unilink is discovered to have gone from *ok_sync* state to *broken* state, a fresh status report is generated at once and broadcast. This is done so that a site that crashes is quickly detected to have done so.

The link state broadcast is transmitted through the network as a high-priority message using the technique of *flooding*. Consider a site i which initiates a link report broadcast at a local time t. Suppose there exists a path of n unilinks from site i to site j. Assume that at time $t-\Delta$, all the sites

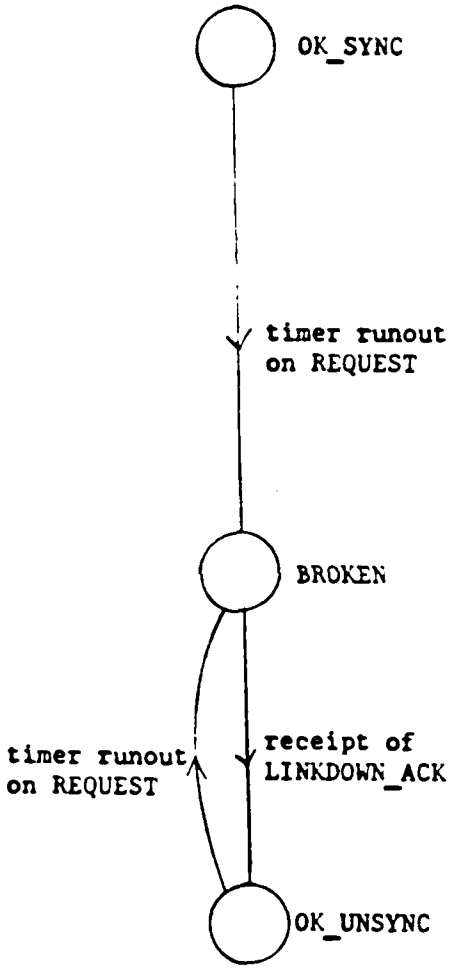


FIG. 2.5. UNILINK STATE TRANSITIONS IN WHICH THE LINK MONITOR MODULE IS INVOLVED.

on that path are in *pause* or *operational* state and the unilinks are in *ok_sync* state. Consider the subpath of this path, r hops long ($1 \leq r \leq n$), excluding site i but including the other end site. If for all r, in the subpath of length r, all the unilinks continue in *ok_sync* state, and all the sites continue in one of the two site states mentioned above till $t + r\Delta$, then the report will reach site j by $t + n\Delta$.

This is achieved by each intermediate site relaying the received report within a period Δ of the *LTR* value, immediately after receipt of the report, for the site from which the report came, on its *ok_sync* unilinks to other sites. This procedure along with adherence to the *CSR* relation guarantees the arrival of link status reports satisfies the time bounds described above.

The state of a unilink is broadcast as *UP* if it is in *ok_sync* state, and as *DOWN* otherwise. How these link status reports are used to update the network status views and to decide if a site should crash itself is described in the next section.

2.2.4.8. The CRASH-OTHERS and CRASH-SELF Modules

Basically, a site i marks another site j or a group of sites containing site j *DOWN* when site i has marked all the unilinks to the site or group of sites *DOWN*. Unless precautions are taken, the site j thus marked *DOWN* may actually be in *operational* state thus violating rule *C*3. For example, it may have been partitioned by physical failure of the bilinks corresponding to the unilinks marked *DOWN* but may not itself have crashed. Further, even the partition may not have actually occurred but the news of some of the unilinks having gone back into *ok_sync* state from *broken* state may not have reached site i in time.

Therefore a site j must have a mechanism by which it can anticipate the possibility of it being marked *DOWN* and cease operation in time and execute a recovery procedure.

To this end, a site j maintains two graphs *CRASH_OTHERS*(j) and *CRASH_SELF*(j). The first is used to detect when site j should mark other sites *DOWN* and the second when it should crash itself. In each graph there is a node for every site in the network. If a bilink connects sites i and j in the network then there are two directed arcs (i,j) and (j,i) in each graph for the corresponding unilinks. In each graph, for each node and each arc there is a *STATE* field and a *TIME* field. When a link state report arrives at a site, it updates its graphs in the manner described below. Note that an update (which consists of a $\{state, time\}$ pair) is effective only if the *time* field in the update is greater than the *TIME* field for the node or arc in the graph being updated, otherwise neither the *STATUS* nor the *TIME* field is changed.

(a) if unilink (p,q) is reported as *DOWN* by site p in a link state report timestamped t then the (*STATE*,*TIME*) fields for arc (p,q) are set to (*DOWN*,t) in both *CRASH_OTHERS*(i) and *CRASH_SELF*(i).

(b) if unilink (p,q) is reported as *UP* by site p in a link state report timestamped t then the (*STATE*,*TIME*) fields for arc (p,q) are set to (*UP*,t) only in *CRASH_OTHERS*(i).

Before we describe under what conditions a site is marked *DOWN* we introduce some notation. Let $NG=\{V,E\}$ be the undirected graph of the network, i.e. it has a node for every site in the network and there is an arc connecting nodes p and q in NG if there is a bilink between the corresponding sites in the network. A *component* C of NG is a subset $\{V_c,E_c\}$ of NG such that (a) every node in V_c is reachable from any other node in V_c through a

path in NG containing only nodes in V_c and (b) E_c consists of all arcs in E which connect nodes in V_c.

A node in V_c which satisfies the condition that at least one arc exists in NG connecting it to a node not in V_c is called a *boundary* node of C. $B(C)$ is the set of boundary nodes of C. A node not in V_c that satisfies the condition that at least one arc exists in NG connecting it to a node in V_c is called a *neighbor* node of C. $N(C)$ is the set of neighbor nodes of C.

The *testing_arcs* OT_c of component C are the set of directed arcs running from $N(C)$ to $B(C)$ in *CRASH_SELF* or *CRASH_OTHERS*. The *self_testing_arcs* ST_c of component C are the set of directed arcs running from $B(C)$ to $N(C)$ in *CRASH_SELF* or *CRASH_OTHERS* (Fig. 2.6).

The significance of the testing_arcs and the self_testing_arcs of a component is as follows. Assume $|V_c| < \left\lceil \frac{n}{2} \right\rceil$, where n is the number of nodes in NG. When a site j outside C has all the arcs in OT_c marked *DOWN* in *CRASH_OTHERS*(j), it marks all the sites in C as *DOWN*. Let t_{max} be the largest of the *TIME* fields for these arcs in *CRASH_OTHERS*(j). Site j marks every site in C *DOWN* with a *TIME* field greater than or equal to $t_{DN} = t_{max} + |V_c|\Delta$. In order to comply with rule $C3$, every site in C, if it has not really crashed, must crash itself by t_{DN}. It will be shown that every site j' in C will find all the self_testing_arcs of C (or a subcomponent of it) to be marked *DOWN* in *CRASH_SELF*(j') by t_{DN}. This condition is the signal for site j' to crash itself. These preliminary remarks should help in understanding the *CRASH_OTHERS* and *CRASH_SELF* algorithms given below.

The module for marking sites *DOWN* in *CRASH_OTHERS*(j) is invoked whenever a link state report arrives at site j declaring a unilink (p,q) to be

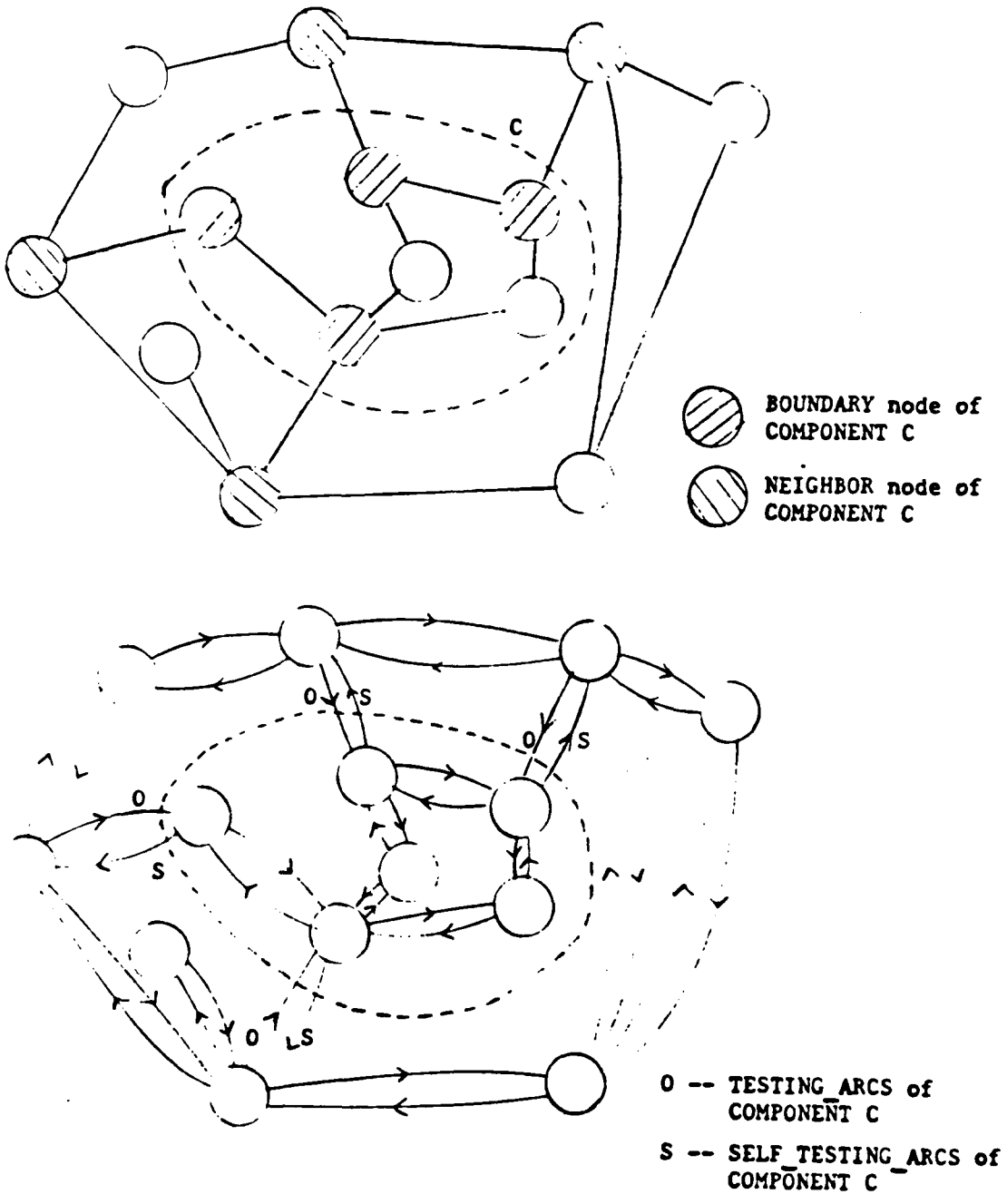


FIG. 2.6. THE NETWORK GRAPH NG AND THE CORRESPONDING CRASH_OTHERS (OR CRASH_SELF) GRAPHS.

DOWN where node q is not already marked *DOWN* in *CRASH_OTHERS*(j). This module executes as described below:

(a) find a component C, if one exists, including node q but not node j such that:

(i) $|V_c| < \left\lceil \frac{n}{2} \right\rceil$, n being the number of nodes in *NG*.

(ii) for all l in OT_c, *STATE*(l)=*DOWN* in *CRASH_OTHERS*(j).

(b) if such a component C is found, let t_{max}=max{*TIME*(l):l *in* OT_c}. Then

(i) bump $C(j)$, if necessary, to a value greater than $t_{DN}=t_{max}+|V_c|\Delta$, not responding to any *LINKUP* or *SITEUP* messages in the interim. (The latter are messages related to recovery procedures to be explained in Section 2.2.4.10).

(ii) mark every node r in V_c not already marked *DOWN* by setting the (*STATE*,*TIME*) fields to (*DOWN*,$C(j)$) in *CRASH_OTHERS* and *CRASH_SELF*.

The module to detect if site j should crash itself is invoked whenever a unilink (p,q) which was *UP* in *CRASH_SELF*(j) is set to *DOWN* as a result of receiving a link state report from p. This module executes as follows:

Find a component C, if one exists including nodes p and j such that

(i) $|V_c| < \left\lceil \frac{n}{2} \right\rceil$

(ii)for all l in ST_c, *STATE*(l)=*DOWN* in *CRASH_SELF*(j).

If such a component exists, enter the *crashed* state.

This procedure must be completed before $C(j)$ exceeds a value Δ beyond the *LTR* value, immediately after receipt of the link status report.

for the neighbor from whom the report was received.

Table 2.1. summarizes the variables stored at site i, for ease of reference.

2.2.4.9. Correctness Arguments (I)

Before stating the recovery procedures for links and sites, it is of interest to show that the above algorithms work if the unilinks which enter *broken* state and sites which enter *crashed* state never leave those states. This will also help in understanding the correctness of the algorithms after recovery procedures have been incorporated.

Thm 1: If site p has site q ($\neq p$) marked as *DOWN* in *CRASH_OTHERS*(p) at local time t, site q is in *crashed* state at time t (i.e. enters *crashed* state before time t).

Proof: Let $t_s \leq t$ be the time when site p marked site q *DOWN* along with the other sites in the component C. For each l in OT_c, let T_l be *TIME*(l) in *CRASH_OTHERS*(p) at the time the sites in component C was found to be suitable for marking *DOWN* and let n_l and b_l be the neighboring and boundary nodes of C to which l is attached.

INDUCTION HYPOTHESIS H1: For each l in OT_c, every site m in C if still *operational* at time $T_l + k\Delta$, $k = 1,2\ldots, |V_c|$, has at least one arc in every path of length k through nodes in C to n_l marked *DOWN* in *CRASH_SELF*(m) by that time.

BASIS: H1 is true for $k=1$ since *CSR* and the link monitor mechanism ensure that the site b_l has l' marked *DOWN* in *CRASH_OTHERS*(b_l) (where l' is the arc running from b_l to n_l) by ($T_l + \Delta$) if it is still *operational* at that time.

Variable Name	Description
$C(i)$	the logical clock value
$LTR(k,i)$	the largest timestamp attached to a message received by site i from site k over the bilink between i and k. k ranges over the neighbors of site i.
$st(i,k)$	the detailed state of unilink (i,k), which may take the values *broken*, *ok_unsync* or *ok_sync*. k ranges over the neighbors of site i.
$state(i)$	the state of site i, which may take the values *crashed*, *sync*, *pause* or *operational*.
$CRASH\_OTHERS(i)$ and $CRASH\_SELF(i)$	directed graphs with a node for each site and 2 arcs for each bilink in the network. Each node and arc has an associated *STATE* and an associated *TIME* field. The *STATE* field takes the values *UP* or *DOWN*.
$issued(i)$	a variable in stable storage set to the value of $C(i)$ every λ ticks.

Table 2.1.: Variables stored at site i.

Assume H1 true for $k = y$ $(y<|V_c|)$.

Consider a node g in C that has a path P of length $y+1$ containing only nodes in C to n_i. The next node h on this path has a path P' of length y to n_i which is P minus the arc from g to h.

By our inductive assumption on H1, site h has an arc on P' marked *DOWN* by $T_i+y\Delta$ if it is still *operational* at that time.

Hence, by $T_i+(y+1)\Delta$, site g has marked *DOWN* either the arc from node g to node h or else the arc in P' which was marked *DOWN* by site h by time $T_i+y\Delta$, since this information would be relayed to site h by $T_i+(y+1)\Delta$.

Hence H1 is true for $k=y+1$ and hence for $k=1,2,..|V_c|$.

But a node in C can only have paths of length at most $|V_c|$ through nodes in C to n_i for all l in OT_c.

Hence there exists a component C' containing q such that $V_{c'}$ is a subset of V_c and for all u in $ST_{c'}$, $STATE(u)=DOWN$ in $CRASH\_SELF(p)$ before $\max\{T_l \, l \text{ in } OT_c\}+|V_c|\Delta \le t_z$. Hence site q will crash itself before $t_z \le t$.

–

The next theorem gives a sufficient condition under which a site will not have to crash itself. We define a component C as *working* in the time interval (t_1,t_2) if:

(i) $|V_c| > \left\lceil \frac{n}{2} \right\rceil$

(ii) Sites corresponding to nodes in C are *operational* at t_1 and suffer no hardware or software failures resulting in their crashing in the given time

interval.

(iii) There exists a subset of E_c, E', such that

(a) the graph (V_c, E') is connected.

(b) for each arc in E', both unilinks corresponding to the arc are recorded as in *ok_sync* state at t_1, and neither suffers any physical failure over the given time interval.

Thm 2: If a component C works over (t_1, t_2), no site in C has to crash itself in this time interval.

Proof: If possible, let one or more sites in C crash themselves in this interval. Let p be the site in C which is the earliest to crash itself. Let C^* be the component which fulfilled the requirements for p to crash itself. Since $|V_{c^*}| < \left\lfloor \frac{n}{2} \right\rfloor$, there must exist a node $q \neq p$ in C which is also in $N(C^*)$ and a path consisting of nodes and arcs in (V_c, E') to node q. Let r be the node in this path in $B(C^*)$. Since p is the first site in C to crash itself, site q is still *operational* at the time p crashes itself. Hence the unilink (r,q) cannot have been marked *DOWN* as a result of q going into *crashed* state and site r reporting the unilink to it as *DOWN* consequently. Also the bilink corresponding to this unilink suffers no physical failure in the given interval. Thus there is no sequence of events that could cause this unilink to be marked *DOWN* in *CRASH_SELF*(p). Thus C^* does not fulfill the conditions for site p to crash itself, contradicting our assumption.

–

2.2.4.10. Recovery Procedures

2.2.4.10.1. Overview

In this section, we describe the recovery procedures for sites and links. The procedures executed by a site as it recovers from *crashed* state through *sync* and *pause* states to *operational* state are described and the recovery procedures for links are described in this context since link recovery is part of site recovery.

In order to motivate the rest of this section, we first briefly summarize the recovery procedure as executed by site i.

In the *crashed* state, the site i sets its clock to a value greater than it ever had hitherto. For this purpose, it makes use of a variable called *issued*(i) kept in stable storage, which is always maintained at most λ behind the clock value.

In the *sync* state, the site synchronizes one or more of its bilinks, i.e., its clock is brought within Δ of the clocks of the corresponding neighbors. Next it appoints one or more neighbors as *guards* to ensure that broadcasts, occurring in the network from now on till it enters *operational* state, reach it even though it may be recorded as *DOWN* by the broadcasting sites in this interval.

In the *pause* state, the site broadcasts news of the recovery of one or more unilinks (i,k) through *LINKUP* messages. When all the sites that are maintaining *CRASH_OTHERS* graphs at the time acknowledge that they have marked the unilink (k,i) *UP* in their *CRASH_OTHERS* graphs, site i, through a *LINKSAFE* message, broadcasts the information that they may now mark unilink (i,k) *UP* in their *CRASH_SELF* graphs.

Here a complication crops up. A unilink l by failing at a critical time may delay the receipt of unilink status reports which would have caused a site j to crash itself. In the case of no recoveries considered earlier, this posed no problem. As we found in the proof of Theorem 1, the failed unilink l, by being itself marked *DOWN* at site j effectively 'substituted' for unilinks, the news of whose failure is delayed in reaching j as a result of l's failure. Hence, site j still crashed itself in time. In the environment we are considering now, in which recoveries do occur, l's failure may cause a delay in reports reaching j but l may then recover and be marked *UP* in *CRASH_SELF*(j), thus ending the substitution. Hence site j may not crash itself when it should have.

For this reason, when site i collects acknowledgements for the *LINKUP* message for a unilink (i,k), it ascertains which unilinks are marked *DOWN* in *CRASH_SELF* graphs in the network (and news of whose failures may have been delayed in reaching sites as a result of the failure of (i,k)). In the subsequent *LINKSAFE* broadcast, the identities of these unilinks are included, and every site on receiving the *LINKSAFE* marks them *DOWN* in its *CRASH_SELF* graph. Thus, when a substituting link is marked *UP* in the *CRASH_SELF* graph of a site, that site also receives the information regarding failures of unilinks which was delayed in reaching it as a result of the failure of the substituting unilink.

We return to the sequence of site i's recovery actions. The site i broadcasts news of its impending transition into *operational* state with a *SITEUP* message. After collecting acknowledgements from all sites maintaining *CRASH_OTHERS* graphs that they have marked site i *UP*, it discards its guards and enters *operational* state. At this point, execution of higher-level

software, which makes use of the status maintenance scheme, may be started.

We now give a detailed description of the above steps.

2.2.4.10.2. crashed → sync

In the *crashed* state, the recovery procedure consists in setting the local clock to a value greater than any it ever had before (ensuring the monotonicity of the clock through crashes) and to wait till at least one neighboring site in *operational* state is discovered.

Each site i has a variable *issued*(i) in *stable storage* [LAM 76]. (Writes to stable storage are atomic and the contents persist through a crash.) On recovery from *crashed* state, the site i adds λ to *issued*(i), sets $C(i)$ to this value and sets *issued*(i) to this new value, too. From then on, *issued*(i) is updated every λ ticks of $C(i)$ to the new value of $C(i)$.

Next, it sets its attached unilinks to *broken* and activates the link monitor. It then waits till at least one of the unilinks directed from it is set to *ok_unsync* state by the link monitor. It periodically sends out a *STATUS_REQ* message on all the *ok_unsync* links. (This message is not timestamped. Timestamped messages are sent out on *ok_sync* unilinks only.) A site j in $N(i)$ responds to this message only if it is in *operational* state. When at least one site in $N(i)$ has responded that it is in *operational* state, the site i enters *sync* state.

2.2.4.10.3. sync → pause

In the *sync* state, the recovery procedure consists in synchronizing the local clock with neighboring sites, initializing the *CRASH_OTHERS* and *CRASH_SELF* graphs and appointing *guards* for its upcoming stay in the

pause state.

The logical clock $C(i)$ is now coupled to the local real-time clock. If the first technique described in Section 2.2.4.5 is used, the local real-time clock which may have stopped functioning in the crash must first be resynchronized with other functioning real-time clocks in the network.

The site i initializes its *CRASH_OTHERS* and *CRASH_SELF* graphs to show all the unilinks directed from it as in *DOWN* state and all other sites and all other unilinks in the network as *UP* with the current value of $C(i)$, i.e. the *TIME* fields are set to the current value of $C(i)$.

Next the *SYNC-LINK* module is invoked sequentially on each of the *ok_unsync* unilinks to sites that have signified that they are *operational*. When a *SYNC-LINK* module invocation returns with the corresponding unilink (i,j) in *ok_sync* state, the site i asks site j to be its *guard* during its upcoming stay in the *pause* state. It does this by sending an *ENTERING_PAUSE* message to site j which responds with an *ENTERING_PAUSE_ACK* if still in *operational* state. When site i has appointed one or more guards, it enters *pause* state. If subsequently the unilinks to all its guards go into *broken* state, before site i has completed its stay in *pause* state and entered *operational* state, the site enters *crashed* state again. When a site j receives an *ENTERING_PAUSE* message, it replies with an *ENTERING_PAUSE_ACK* if in *operational* state. From then on, till it

(a) enters *crashed* state itself,or

(b) receives a *LEAVING_PAUSE* message (to be described in the next section) or

(c) the unilink (j,i) leaves *ok_sync* state,

in responding to any *SITEUP*,*LINKUP* or *LINKSAFE* messages (to be dis-

cussed below), site j specifies site i to be in *pause* state in its response. In the last case, site j bumps up its clock by Δ before replying to any of the aforesaid messages, by which time site i will be aware of site j ceasing to be its guard.

The larger the number of guards site i appoints before entering *pause* state, the less likely it is that its recovery while in *pause* state will have to be restarted from *crashed* state as a result of its unilinks to all its guards leaving *ok_sync* state.

2.2.4.10.4. The SYNC_LINK Module

The function of this module is to synchronize the clock of the invoking site with that of the site at the other end of the unilink for which it is invoked, thus setting the unilink (and its reverse counterpart) to *ok_sync* state.

A *MY_TIME_IS* message carrying the current value of C(i) is sent on the unilink (i,j).

If the response (we describe the appropriate responses to messages sent out by this module below) carries a clock value within Δ of the current logical clock value, a *SYNCHED* message is sent on the unilink carrying the current value of $C(i)$. If a timestamped *SYNCH_ACK* message is received and the timestamp is within Δ of the current value of $C(i)$, the state of the link is set to *ok_sync* (the timestamp being used to set the corresponding *LTR* register) and the module returns. If the timestamp on the *SYNCH_ACK* is more than Δ less than the current value of $C(i)$, the *SYNC_LINK* procedure is restarted.

If the response to the *MY_TIME_IS* message carries a clock value exceeding the current value of $C(i)$, by more than Δ, the clock synchronization module is invoked to bump $C(i)$ up to the clock value in the response. Then the *SYNC_LINK* procedure is started again.

If the response to the *MY_TIME_IS* message carries a clock value which is less than the current value of $C(i)$ by more than Δ, the module restarts the *SYNC_LINK* procedure.

If a *SYNCH_NACK* message is received in response to the *SYNCHED* message, it may bear a clock value exceeding that of the *SYNCHED* message by more than Δ. In this case, the same procedure (described above) undertaken when the response to a *MY_TIME_IS* message exceeds the current value of $C(i)$ by more than Δ, is executed. Otherwise, the *SYNC_LINK* procedure is simply restarted.

A site j should respond to the messages of the *SYNC_LINK* module only when in *operational* state.

The response to a *MY_TIME_IS* message from site i when unilink (j,i) is in *ok_unsync* state (the unilink is set to *broken* state if it is not in *ok_unsync* state when this message arrives) is to bump $C(j)$ up to the clock value carried by the message, if necessary. Then a *MY_TIME_IS_ACK* message bearing the current value of $C(j)$ is sent.

The response to a *SYNCHED* message is, if the clock value borne by it is within Δ of $C(j)$, to set the unilink (j,i) to *ok_sync* state using the clock value to set the corresponding *LTR* register and to return a timestamped *SYNCH_ACK* message. If the clock value of the *SYNCHED* message is not within Δ of $C(j)$, a *SYNCH_NACK* message carrying the current value of $C(j)$ is returned.

When a site j in *operational* state sets a unilink (j,i) to *ok_sync* state, it invokes the *LINKUP_BROADCAST* module (described below) on it.

If during the above procedure, the unilink (i,j) gets set to *broken* state, the *SYNC_LINK* module returns at once.

In the *pause* and *operational* states, this module is invoked whenever a unilink goes from *broken* to *ok_unsync* state.

2.2.4.10.5. pause → operational

In the *pause* state, the recovery procedure consists in informing the network of the recovery of the unilinks directed from the site and of the intended transition of the site to *operational* state.

The site i starts issuing and relaying link state reports, and processing them to update its *CRASH_OTHERS* and *CRASH_SELF* graphs as described in Section 2.2.4.8 with the exception that the *CRASH_SELF* module is not invoked for the time being. It responds to any *LINKUP*, *LINKSAFE* and *SITEUP* messages received as described below.

Next, the site i invokes the *BROADCAST_LINKUP* module (described below) in parallel on all *ok_sync* unilinks (i,j) attached to it. These invocations, if successful, will produce broadcasts of *LINKSAFE* messages for these unilinks. (See the description of the *BROADCAST_LINKUP* module below for a description of *LINKSAFE* messages.)

When all the *BROADCAST_LINKUP* module invocations have returned, if not even one of the *ok_sync* unilinks has been marked *UP* in *CRASH_SELF*(i) as a result of receipt of a *LINKSAFE* message, the site re-enters *crashed* state. If at least one of the *ok_sync* unilinks has been marked up in *CRASH_SELF*(i), the *CRASH_SELF* module is invoked. From this instant on,

the *CRASH_SELF* module is invoked whenever a received link state report makes it appropriate to do so as described in Section 2.2.4.8. If when the module returns, the site has not entered *crashed* state, the following procedure is executed.

The site i broadcasts a timestamped *SITEUP* message. (The responses to this message as well as to the LINKUP and LINKSAFE messages should carry the timestamps of the messages to allow them to be matched up.) The appropriate responses to the *SITEUP* message for any site j in the network are:

(i) if in *crashed* or *sync* states, respond with a timestamped *SITEUP_ACK*.
(ii) if in *pause* state or in *operational* state, set the (*STATE*,*TIME*) fields in *CRASH_OTHERS*(j) and also in *CRASH_SELF*(j) for site i to (*UP*,*current local time*), provided the timestamp on this message is greater than the *TIME* fields. If in *operational* state, the ids of all sites that site j is currently guarding should be added to the acknowledgement.

Site i periodically resends timestamped *SITEUP* messages till all sites that have not responded are marked *DOWN* in *CRASH_OTHERS*(i). In addition, if any site k is specified as being in *pause* state by one of its guards, the site and its guards are sent *SITEUP* messages, till either the site responds or its guards cease specifying site k as being in *pause* state (by getting marked *DOWN* themselves in *CRASH_OTHERS*(i) or returning *SITEUP* acknowledgements without specifying node k). In the latter case, either the site k has entered *UP* state, in which case site i will have received a *SITEUP* message from it, or all the guards have stopped being guards before site k enters *operational* state, in which case site k reenters *crashed* state.

The site i next sends a *LEAVING_PAUSE* message to those guards to which its unilinks have not left *ok_sync* state since the time they responded to site i's *ENTERING_PAUSE* message. It waits for a *LEAVING_PAUSE_ACK* or for the corresponding unilink to enter *broken* state. When either of these events has occurred for each unilink over which a *LEAVING_PAUSE* message was sent, it enters *UP* state if at least one *LEAVING_PAUSE_ACK* is received, otherwise it reenters *crashed* state. Only after entering *UP* state is execution of higher-level software resumed.

The original *SITEUP* message is sent by flooding (but without the time constraints imposed on the flooding mechanism by the link state reports); all responses and subsequent *SITEUP* retransmissions can be sent by normally routed messages. Similar considerations hold for the *LINKUP* and *LINKSAFE* messages discussed in the next section.

2.2.4.10.6. The BROADCAST_LINKUP module

The function of this module, when invoked on the unilink (i,j) is to ensure that the *CRASH_OTHERS* graphs in the network have been updated to show the unilink (j,i) *UP* and then to have the unilink (i,j) marked *UP* in *CRASH_SELF* graphs.

When a link (i,j) is restored from *broken* to *ok_sync* state, this information is broadcast by the link state reporting mechanism described in Section 2.2.4.7. and other sites appropriately update their *CRASH_OTHERS* graphs. However, the updating of the link state in the *CRASH_SELF* graphs must be postponed till it is certain that all other sites have updated their *CRASH_OTHERS* graphs, in order to leave no chance of rule *C*3 being violated.

Hence the *BROADCAST_LINKUP* module which is responsible for getting unilinks marked *UP* in the *CRASH_SELF* graphs, does this job in two phases.

Phase I: The site i, which is executing the *BROADCAST_LINKUP* module for unilink (i,j) broadcasts a timestamped *LINKUP* message carrying the *LTR* value, say t_L, for node j. The responses to this message from any node k are:

(i) if in *crashed* or *sync* state, return a timestamped *LINKUP_ACK* specifying site k's current state.
(ii) if in *pause* state, the (*STATE*,*TIME*) fields for unilink (j,i) are set to (UP,t_L), in *CRASH_OTHERS*(k). A timestamped *LINKUP_ACK* is returned. The identities of unilinks directed from site k which are marked *DOWN* in *CRASH_SELF*(k) are specified in the *LINKUP_ACK* along with their *TIME* fields in the same graph.
(iii) If site k is in *operational* state, the unilink (j,i) is set to (UP,t_L) in *CRASH_OTHERS*(k). A timestamped *LINKUP_ACK* is returned. The identities of those unilinks directed from node k which are marked *DOWN* in *CRASH_SELF*(k) are sent along with their *TIME* fields from the same graph in this acknowledgement. The ids of all guarded sites should also be specified.

Timestamped *LINKUP* messages carrying a time t_L are resent till all sites not marked *DOWN* respond. Responses from sites specified by their guards are also sought. For every site, whether specified as guarded or not, from whom an explicit response is not received, the following procedure is applied.

Let r be a site marked *DOWN* in *CRASH_OTHERS*(i) from which a response is not collected. Let $t-\Delta$ be the maximum of the *TIME* fields for those neighbors of site r marked *DOWN* in *CRASH_OTHERS*(i). Then a *LINKUP* message timestamped $\geq t$ must be sent to all the *UP* neighbors of site r, if any and their responses collected. (If any of these *UP* sites get marked *DOWN* before responding, the procedure should be restarted with these neighbors newly marked *DOWN* included in the set of neighbors for whom the maximum *TIME* field is computed.) If none of the *UP* neighbors specify r as a guarded site in their response, site r if still marked *DOWN* either has not appointed any guardians at time t or has reentered *crashed* state as a result of all unilinks to its guardians leaving *ok_sync* state. Hence site r is taken to have implicitly responded that it is in *crashed* or *sync* state with a message timestamped t.

The responses are processed as follows:

(i) A response, implicit or explicit, from a site in *crashed* or *sync* state is treated as specifying that all the unilinks directed from that node should be marked *DOWN* in *CRASH_SELF*(i) at the time corresponding to the response timestamp.

(ii) for each unilink specified as *DOWN* at a time t in responses from sites in other states, the (*STATE*,*TIME*) fields for that unilink are set to (*DOWN*,t) in *CRASH_SELF*(i).

The site i then enters Phase II.

Phase II: The site i broadcasts a *LINKSAFE* message for unilink (i,j), carrying the time t_L. This message carries in addition, a list of all the unilinks marked *DOWN* in *CRASH_SELF*(i) along with their *TIME* fields from this

graph. The responses to this message from a site k are:

(i) if site k is in *crashed* or *sync* state, return a timestamped *LINKSAFE_ACK*.

(ii) if k is in *pause* or *operational* state, set the (*STATE*,*TIME*) fields for unilink (i,j) to (UP,t_L) in *CRASH_SELF*(k). For each unilink specified as *DOWN* at a time t in the *LINKSAFE* message, set the (*STATE*,*TIME*) fields for that unilink to *DOWN* at the time t in *CRASH_SELF*(k). A timestamped *LINKSAFE_ACK* is returned. If in *operational* state, the ids of all guarded sites should be specified in the acknowledgement.

The *LINKSAFE* message should be resent till all sites not marked *DOWN* respond. Responses from guarded sites are collected as for *SITEUP* messages. Then the module returns.

The module returns immediately if the unilink (i,j) goes to *broken* state in Phase I or Phase II.

2.2.4.11. Correctness Arguments (II)

In this section, we develop analogs to Theorems 1 and 2 for our scheme with recovery procedures incorporated.

In proving the analog of Theorem 1, we have to show that recoveries of unilinks and sites do not prevent a site from crashing itself in time when needed if it has been marked *DOWN* at some other site.

Thm 3: If site p has site $q \neq p$ marked *DOWN* in *CRASH_OTHERS*(p) at local time t, site q is not in *operational* state at time t.

Proof: Arrange the various events corresponding to marking *DOWN* of sites in increasing order of the local times at which they occur (if some occur at the

same time, arrange according to increasing id of the site at which the event occurs).

Let t_j be the logical time of the jth such event.

INDUCTION HYPOTHESIS H1: If site p has site $q \neq p$ marked *DOWN* in *CRASH_OTHERS*(p) at local time $t < t_j$, site q is not in *operational* state at time t.

BASIS: Obviously true for $j=1$, since no site marks any other node *DOWN* before t_1.

Assume H1 true for $j=x$.

Consider the marking *DOWN* event at t_x, which, say, is the marking *DOWN* at site m of sites in component C. For each l in OT_c, let T_l be *TIME*(l) in *CRASH_OTHERS*(m) at the time the component C was found to be suitable for marking *DOWN* and n_l be the node in $N(C)$ to which l is attached. Let $t_{max} = \max \{T_l\}$ for l in OT_c so that $t_x \geq t_{max} + |V_c| \Delta$.

Consider a node $a = r_k$ in C which has a path $k \leq |V_c|$ hops long to $n_l = r_0$ (the path being $r_k, r_{k-1}, \ldots, r_1, r_0$). We will show that at t_x, site a has at least one unilink on the path from a to n_l, marked as *DOWN* in *CRASH_SELF*(a).

Define a series of local times $\vartheta_0, \vartheta_1, \ldots \vartheta_k$ for nodes $r_0, r_1, \ldots r_k$ as follows:

(i) ϑ_0 is defined as T_l.

(ii) If the report by site r_0 at T_l of unilink (r_0, r_1) being *DOWN* was due to physical failure of the bilink between r_0 and r_1, and if node r_1 is in *pause* or *operational* state before $\vartheta_0 + \Delta$ for a sufficiently long time, its link monitor will detect this failure and initiate a link state report. Let ϑ_1 be the latest time

before $\vartheta_0+\Delta$ when the monitor initiates a broadcast and marks the unilink (r_1,r_0) *DOWN* in *CRASH_SELF*(r_1), if this situation exists. On the other hand, r_1 may not be in either of the above states during the period of the failure. It is also possible that the report at T_i by r_0 may be caused by a crash of r_1, which makes the unilink from r_0 to r_1 enter *broken* state. In these cases, where the link monitor in r_1 does not initiate any broadcast of the failure which caused the report at T_i, we define ϑ_1 as the latest time before $\vartheta_0+\Delta$ when r_1 enters *crashed* state.

(iii) For $1<i\leq k$, ϑ_i is defined as follows:

(a) if ϑ_{i-1} is the time of marking *DOWN* in *CRASH_OTHERS*(r_{i-1}) and *CRASH_SELF*(r_{i-1}) of some unilink on the path from r_{i-1} to r_0, as a result of a link state report initiated by one of the sites $r_1,r_2,\ldots,r_{i-1}$ and if this report reaches r_i by $\vartheta_{i-1}+\Delta$, ϑ_i is defined as the time at which this unilink is marked *DOWN* in *CRASH_SELF*(r_i) as a result of receiving this report.

(b) However, this report may not reach r_i by $\vartheta_{i-1}+\Delta$ because of the failure of the bilink between r_{i-1} and r_i. Alternatively, r_{i-1} may have entered *crashed* state at ϑ_{i-1}. In these cases, we define ϑ_i as the latest time before $\vartheta_{i-1}+\Delta$ when the link monitor in r_i detected the *broken* or *ok_unsync* state of the unilink from r_i to r_{i-1} and initiated a link state report causing this unilink to be marked *DOWN* in *CRASH_SELF*(r_i).

(c) Lastly, if site r_i was not in *operational* or *pause* state at a time before $\vartheta_{i-1}+\Delta$ to make any of the above situations occur, we define ϑ_i as the latest time before $\vartheta_{i-1}+\Delta$ when site r_i enters *crashed* state.

INDUCTION HYPOTHESIS H2:

(i) $\vartheta_i \leq T_i + i\Delta$

(ii) Site r_i has some unilink on the path to r_0 in front of it marked *DOWN* in *CRASH_SELF*(r_i) from ϑ_i to t_x at every instant it is maintaining this graph.

BASIS: The first part of H2 is true for $i=1$ because $\vartheta_1 \leq T_1 + \Delta$ by definition. The second part is true for the following reason. Since r_0 has reported the unilink (r_0, r_1) as *DOWN* at T_1, the unilink (r_1, r_0) cannot become *ok_sync* before T_1. Hence the *LINKUP* broadcast cannot be started before T_1. But site m, which marks the nodes in *C DOWN* at t_x, cannot respond to this broadcast, whether in *pause* or *operational* state, before t_x. Further, since site m has entered *pause* state before T_1 site r_1 cannot avoid waiting till this response is received from site m whichever state, *pause* or *operational* it is in. (Remember that H1 is assumed to be true for $j=x$, hence site r_1 cannot incorrectly consider sites in *operational* state, including those that might be guarding m, to be *DOWN*). Therefore, the corresponding *LINKSAFE* message can be broadcast only after t_x by r_1. Hence the unilink (r_1, r_0) remains marked *DOWN* in *CRASH_SELF*(r_1) from ϑ_1 to t_x at any instant that r_1 is maintaining this graph in this period.

Assume H2 true for $i<y$.

Consider r_y which either crashed at ϑ_y or marked a unilink in front of it on the path to r_0, *DOWN* at ϑ_y. In the first case, after recovery, the unilink from r_y to r_{y-1} cannot become *ok_sync* before ϑ_{y-1}. Hence the *LINKUP* broadcast for this unilink cannot be started before ϑ_{y-1}. In the second case, the marking *DOWN* is the result of a link status report

issued at ϑ_d by r_d for some $d \leq y$, reporting the unilink (r_d, r_{d-1}) to be *DOWN*. In this case, the *LINKUP* broadcast for the unilink (r_d, r_{d-1}) cannot begin before ϑ_{d-1}. In either case, it follows from our inductive assumption on H2 and the way the responses to the *LINKUP* broadcast are processed, that if these responses are received before t_s, the ensuing *LINKSAFE* will indicate some unilink (r_s, r_{s-1}) on the path to r_0 in front of the unilink whose safety is being broadcast, as to be marked $(DOWN, t)$ in the *CRASH_SELF* graph , where $\vartheta_s \leq t \leq t_s$, of every site receiving the *LINKSAFE* message.

Hence the marking *UP* in *CRASH_SELF*(r_y) of the unilink in front of r_y marked *DOWN* by it at ϑ_y, or the marking *UP* of the link (r_y, r_{y-1}) on recovery if it crashed at ϑ_y will be accompanied by the marking *DOWN* of some unilink in front of the unilink being marked *UP*, if this marking *UP* occurs before t_s. If, in turn, this unilink is marked *UP* before t_s, some other unilink in front of it on the path to r_0 will be marked *DOWN*. [Ultimately, the unilink (r_1, r_0) may be marked *DOWN* and the *LINKSAFE* for this unilink cannot be issued, as already indicated, before t_s.] Hence site r_y will have some unilink in the path to r_0 marked *DOWN* in *CRASH_SELF*(r_y) from ϑ_y to t_s. Moreover, since $\vartheta_y \leq \vartheta_{y-1} + \Delta$ by definition and since $\vartheta_{y-1} \leq T_i + (y-1)\Delta$ by our inductive assumption on H2, it follows that $\vartheta_y \leq T_i + y\Delta$.

Hence, H2 is true for i=y and hence for i=1,2,...k.

Thus at t_s, the site a has at least one unilink on every path from itself to n_l marked *DOWN* in *CRASH_SELF*(a) for all l in OT_c.

Hence as in Theorem 1, no site in C will be in *operational* state at t_s.

Thus at t_x, the *CRASH_OTHERS* graph of every site in *operational* or *pause* state is correct in that it shows no site as *DOWN* that is actually in *operational* at that time.

After t_x, no sites are marked *DOWN* in any *CRASH_OTHERS* graph till t_{x+1} when the next marking *DOWN* of a site or sites occurs. Hence to show that the graphs remain correct in this period, it suffices to show that no site enters *operational* state before informing any site that has entered *pause* or *operational* state and marked it *DOWN* that it is entering *operational* state, through a *SITEUP* message.

To show this, we order the events corresponding to sites entering *operational* state in the above interval in increasing order of times that they enter this state. Consider the first such recovery, say of site w. Site w's *CRASH_OTHERS* graph was correct at t_x and is correct at all instants to the instant it enters *operational* state, since it is the first site to enter *operational* state after t_x. Since a site can fail to have itself marked *UP* at sites that have marked it *DOWN*, whether they have done so when they were in *pause* state or in the *operational* state, only by incorrectly considering sites in *operational* state *DOWN*, it follows that site w does get itself marked *UP* at all appropriate sites before it enters *operational* state. Hence all the *CRASH_OTHERS* graphs are correct at the time of the first entry into *operational* state after t_x, and using the same arguments, at every subsequent entry into *operational* state thereafter till t_{x+1}.

Hence H1 is true for $j = x+1$, and therefore for all j, proving the theorem.

Before stating the analog of Theorem 2 for the case where recovery of links and sites does occur, we introduce some notation. A unilink (i,j) is ***safe*** at time t, if it is marked *UP* in both graphs with *TIME* field values after which it has not left *ok_sync* state till time t at all sites in *pause* or *operational* state, i.e. it is safe from the time a *LINKSAFE* broadcast for it has completed, till it suffers the first failure after the initiation of the corresponding *BROADCAST_LINKUP*. A ***dynamic component*** $C(t)$ of $NG=\{V,E\}$ is a time-varying graph $\{V_c(t),E'(t)\}$, such that $V_c(t)$ is a subset of V and $E'(t)$ is a subset of $E_c(t)$, the set of arcs in NG which connect nodes in $V_c(t)$, such that $\{V_c(t),E'(t)\}$ is connected for all t. Thus nodes and unilinks enter $C(t)$, stay for periods of time called ***membership periods*** and then leave.

A dynamic component is ***safe*** during the period (t_1,t_2), if

(i) each site in the component is *operational* at the beginning of each of its membership periods in this interval and suffers no crashes due to hardware or software failures in the membership period,

(ii) if each unilink in the component is safe at the beginning of each membership period and suffers no physical failures during the membership period, and

(iii) if $|V_c(t)| > \left\lceil \frac{n}{2} \right\rceil$ for all t in the given interval.

Thm 4: If a dynamic component $C(t)$ is safe during (t_1,t_2), no site is forced to crash itself during any of its membership periods in this interval.

Proof: If possible, let one or more sites in $C(t)$ crash themselves during their membership periods in this interval. Let p be the site in C which is the earliest to crash itself in this interval during one of its membership periods, say

at time t. Let C^* be the component that satisfies the conditions of the self-crash procedure which p finds in its *CRASH_SELF* graph. Since $|V_{c^*}| < \left\lceil \frac{n}{2} \right\rceil$, there must exist a node $q \neq p$ in $C(t)$ which is also in $N(C^*)$, and a path of nodes and arcs in $\{V_c(t), E(t)\}$ to node q. Let r be the node in this path in $B(C^*)$. Since p is the first site in $C(t)$ to crash itself in the given interval during one of its membership periods, sites q and r are still *operational* at t. Hence the unilink (r,q) which was safe at the beginning of its current membership period, cannot have been marked *DOWN* in *CRASH_SELF*(p) either because r crashed and therefore a *LINKSAFE* message was able to specify the unilink as *DOWN* or because q crashed and site r reported the unilink *DOWN* subsequently. Further, the bilink corresponding to this unilink suffers no physical failure in its current membership period. Hence there exists no sequence of events that could have caused unilink (r,q) to be *DOWN* in *CRASH_SELF*(p) at time t, contradicting our assumption.

—

2.2.5. Overhead Considerations and Choice of Parameters

In the Arpanet, link state reports are broadcast from every site every 1 minute or so for routing purposes and broadcast propagation times are less than 1 second (typically 100ms) [MCQ 80].

Assuming that the networks under consideration have similar size and communication bandwidth, we can choose the period of broadcast, τ_L = 1 minute so that the communication overhead from the link state reports, which in any case are needed along with other information for routing purposes, is of the same order. When no link or site failures occur, the only

additional communication overhead from our scheme arises from messages for clock synchronization and for link monitoring. For every unilink recovery, the messages required are (i) the *LINKUP* and *LINKSAFE* broadcasts and (ii) an acknowledgement from each site to the broadcaster for each of the two broadcasts. As mentioned before, the broadcasting is done by flooding. In our algorithm, as presented, when a bilink recovers, the communication costs will correspond to two unilink recoveries. When a site with L attached bilinks recovers from a crash, the communication costs will correspond to $2L$ unilink recoveries plus a *SITEUP* broadcast and its acknowledgements. Optimizations in which the messages are piggy-backed should be straightforward but are not explored in this thesis. Even if the *LINKUP* broadcasts for the $2L$ unilink recoveries are not piggybacked, they can be performed in parallel. The same is true for the *LINKSAFE* broadcasts. Hence, the time required from the instant a site recovers physically to the instant it enters *operational* state is the time required for 3 sequential broadcasts (*LINKUP*, *LINKSAFE* and *SITEUP*) and their acknowledgements. Δ is chosen so that sending a timestamped message every Δ interval to each neighbor is not a burden. Even this burden is absent if other timestamped messages concerned with normal processing are being exchanged. Further, the choice should give the site sufficient flexibility in its schedule for flooding link state broadcast messages. Δ= 10 seconds is a reasonable choice. τ_c the real clock timer can be set to 1~2 seconds without consuming an appreciable amount of site resources in updating the local clock. λ, the interval between stable storage writes can be chosen as ~30 seconds without using up much disk bandwidth.

2.3. An Algorithm for Multiple Copy Updating

2.3.1. Introduction

In this section, we construct an algorithm to update a replicated file in a point-to-point network, using the status maintenance scheme described in the previous section. The algorithm can be briefly characterized as follows:

(i) File copies that are used in executing commands from transactions can be in either of two states, called the HOT and WARM states.

(ii) When an update command is executed, the HOT copies are (atomically) updated immediately. Periodically, the HOT sites bring the WARM sites up-to-date by sending them a list of updates, accumulated since the last time the WARM sites were brought up-to-date. Thus the HOT copies represent the latest version of the file at all times.

(iii) A read command is directed to a HOT site if the current version of the file is required. If it is not essential to obtain the current version, the command may be directed to a WARM site.

(iii) If the set of HOT sites is depleted due to site crashes, one or more WARM sites joins the set as necessary. In order to detect such a depletion when it occurs, the status maintenance scheme described in the previous section is used.

2.3.2. Previous Work in Updating Replicated Files

A file is replicated

(i) in order that its availability may be preserved in the face of failures.

(ii) in order to reduce the response time for read access. If a file copy exists

at the site of access or near it, the response time is less. The factors which constrain the degree of replication are storage costs, update execution costs and the response time for updates.

The costs involved in processing an update result from startup (parsing the command, authorization checking, setting up the necessary processes and communication ports), concurrency control (obtaining the required locks), retrieval of appropriate data from storage, computing the new values and writing them back to storage, and commit processing.

The response time for an update increases with the number of sites that must be accessed before a 'done' can be returned to the originator of the transaction. For example, in the 'performance' algorithms of distributed INGRES [STO 79], only one specially designated copy, called the *primary*, is updated before the 'done' is signaled. If most of the update transactions originate at the primary, then the response time for most update transactions will be similar to that obtained if the file existed only at the site of origin of the transaction. However, if the primary fails before relaying the update to the remaining copies, the update is 'lost' which can result in a catastrophe. Therefore, in the 'reliability' algorithms of distributed INGRES, all copies of the file are updated atomically and then a 'done' is signaled. This results in higher communication costs, a higher load on the resources of the sites holding copies, as will as a higher response time.

Other schemes [THO 76, GIF 79] have been proposed in which there is no designated HOT set of copies, representing the latest version of the file, at a given time. Rather the HOT set may 'float' from update to update even when there are no site or link failures. In these schemes, a majority of sites holding copies (or a set of sites holding a majority of votes between them in

Gifford's weighted voting scheme [GIF 79]) must be accessed before any update can complete, hence the costs per update and response time increase with the number of copies.

In our solution, the response time and immediate costs per update do not increase if the number of WARM copies is increased. Moreover, the updating of the WARM copies can be scheduled when there is surplus capacity in the sites involved and in the communication system. Further, there is no commit processing in updating WARM copies and the storage accessing sequence for installing a batch of updates can be more efficient than if they are installed separately. Hence the costs for the deferred updating of a WARM copy are less than for the immediate updating of a HOT copy.

The disadvantage resulting from the deferred update is that reading a WARM copy may not give the latest version of the file. However, in many cases, the latest version may not be required for a read access. For example, in a banking application, if a customer has just made a withdrawal of funds and makes a subsequent query about his/her balance, the withdrawal should be reflected in the value returned in answer to the query. Hence a HOT copy should be used to answer the query. But a transaction computing the sum-total of balances of all customers of the bank will not require the latest value of each balance, and can make use of a WARM copy of this information. In other situations, if an old version is obtained, it will be detected as not current and a fresh read initiated. File catalogs which store the whereabouts of files in the network are an example of such a case.

An important component of any scheme for managing replication is the method of recovery. In our algorithm, a recovering site first obtains a WARM copy and then obtains the additional updates (if any) to make it HOT if and

when the time comes for it to join the set of sites with HOT copies. If a HOT copy were used right away, the file would be locked out to update access for the entire period the recovering site is accessing the HOT copy. The algorithms of [STO 79] and [BER 80] make use of a reliable queuing mechanism to buffer updates for crashed sites. This queuing mechanism achieves reliability by storing the queued messages at multiple sites in the network. As can be seen from [HAM 80], the design of such a mechanism can be quite complex. Besides, if a site is down for a long time, the number of queued messages for it may be so large that storing them in the above manner may be infeasible. In [GIF 79] on the other hand, a recovering site places a read lock on the file to obtain a HOT copy, thus preventing update access for the duration this process is occurring, which may be appreciably long in a long-haul network. In our algorithm, the recovering site obtains a WARM copy from a site which has a WARM copy if possible and thus avoids locking out update access on the HOT copies. Only when a site with a WARM copy is entering the set of sites with HOT copies is a read lock placed on a HOT copy. Update access to the file is prevented while the lock is held. However, if the refreshing interval for the sites with WARM copies is properly chosen, the time for obtaining the additional updates that have occurred since the last refresh will be small compared to the time required to transfer the entire file, hence the locking out period will also be comparatively small.

2.3.3. States and State Transitions

The complete state diagram for a site holding a copy of the file is shown in Fig. 2.7. In the sequel, we will refer to sites in state S as S sites where S may be DEAD,COLD,WARM,HOT or PRIMARY.

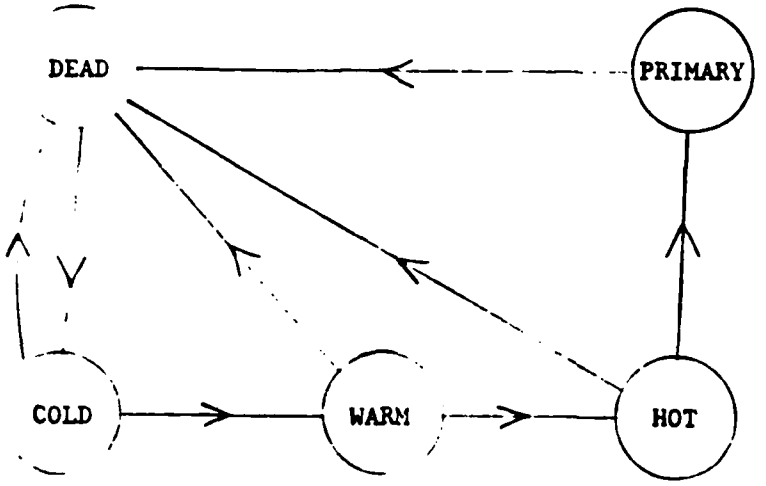


FIG. 2.7. STATE DIAGRAM FOR A SITE CARRYING A FILE COPY.

A site is in DEAD state if it is down or if it has recovered but not yet initiated the execution of software for file recovery. It is in COLD state from the moment of initiation till it has obtained a WARM copy of the file. It enters WARM state on obtaining a WARM copy. While in the WARM state, the site services read requests that do not necessarily require the latest version of the file. If it does not crash, eventually it enters the head of the queue of WARM sites waiting to join the set of HOT sites. When the next failure of a HOT site occurs, it makes itself completely up-to-date and enters HOT state. While in the HOT state, the file copy at the site is updated atomically with other HOT copies when an update request is received. In addition, the site also services read requests, which thereby obtain the latest version of the file. The number of HOT copies is maintained at P. If the set of sites holding HOT copies falls below P in strength, update requests are not accepted till the set regains its full strength. This is done to maintain the probability of 'losing' the latest version of the file below a given level determined by the value of P. One of the sites holding a HOT copy is designated as the PRIMARY. The PRIMARY performs two tasks in addition to those done by a member of the set of HOT sites. First, it is responsible for broadcasting lists of accumulated updates periodically to the WARM sites. Second, when the strength of the set of HOT copies falls below P, it is responsible for helping the WARM sites, which join the HOT set as a result, to make their copies HOT. A site in HOT state enters the PRIMARY state, when all the sites that were in the HOT or PRIMARY states when it entered the set have crashed. In order to have at most one PRIMARY and at most P-1 HOT sites at a given time, the sites holding copies of the file form themselves in a queue which determines their priority for entering the HOT state or becoming PRIMARY (Fig. 2.8). This queue is ordered in increasing order of the times on the global clock that the sites

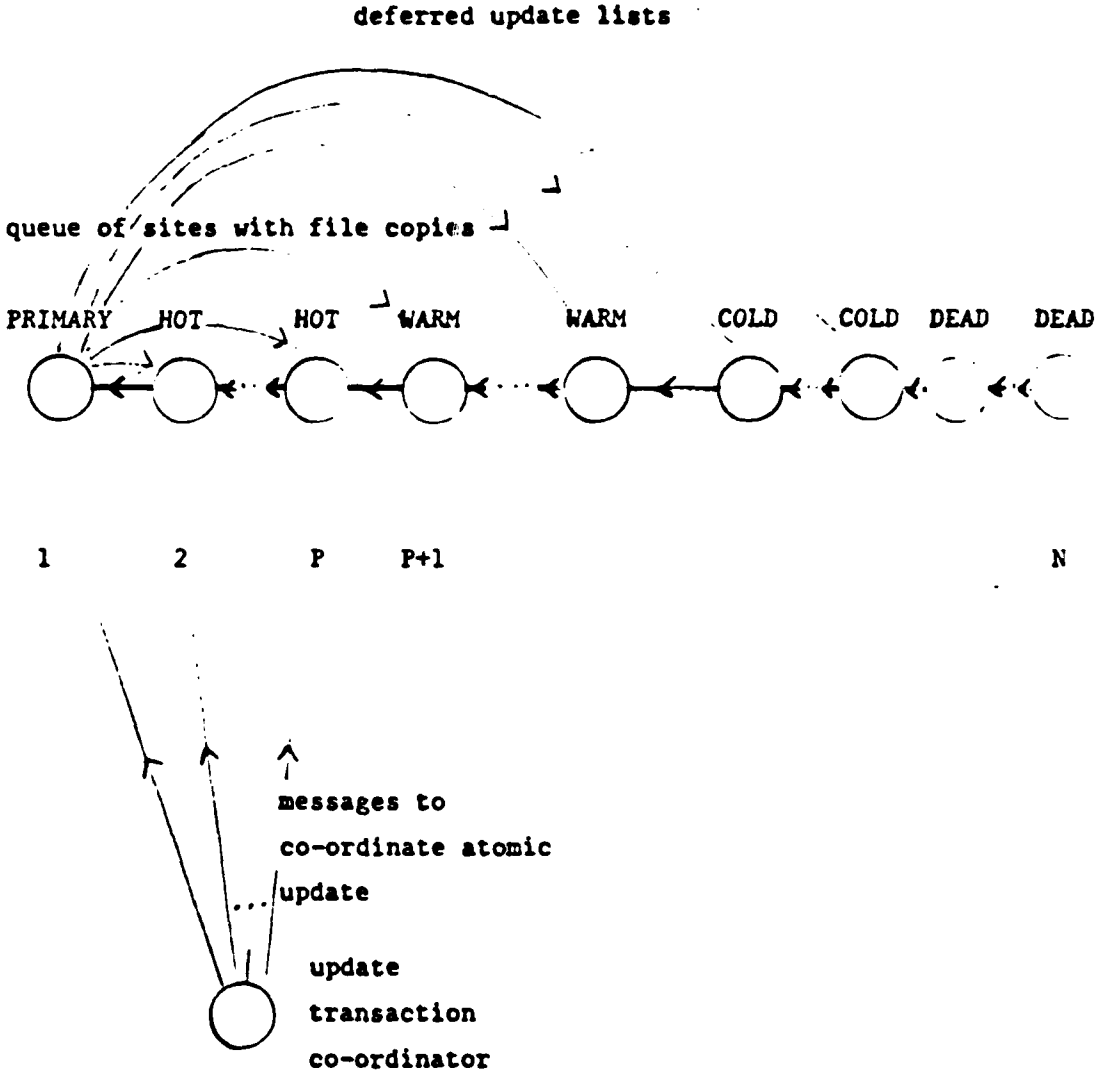


FIG. 2.8. CONFIGURATION OF SITES FOR UPDATING THE REPLICATED FILE.

not in DEAD state entered the COLD state. The DEAD sites are at the rear of the queue in arbitrary order. The queue is maintained at each site not in DEAD state. The global clock allows these queues to be maintained in a consistent yet autonomous manner, i.e. the queues are not updated atomically, but still they permit the sites to regulate their entry into the HOT or PRIMARY states on the basis of the local queue while maintaining the constraint on the number of sites in the HOT state and the requirement of a single PRIMARY. Previous algorithms for selecting primaries e.g. those in [STO 79,GAR 82] rely on some form of atomic updating of status information. Therefore, they suffer from complications which arise if sites crash or recover during the atomic update.

2.3.4. The ADA Multitasking Facility and Remote Procedure Calls

In the appendix of this chapter, we specify our algorithm in ADA. The program displayed in the appendix does not represent an existing implementation of the algorithm, but is intended to be a more formal specification than the informal description given in Section 2.3.6. In this section, we outline the multitasking facility of ADA and a remote procedure call mechanism adequate for the problem at hand.

Consider the example task READER_WRITER taken from the ADA Reference Manual [HON 79] (Fig. 2.9). The *procedure* READ and the *entry* WRITE in the task declaration at the top can be called by other tasks. The entries START and STOP declared in the task body can only be called within the task body itself. The procedure READ can be executed on behalf of several tasks simultaneously. But the procedure entry WRITE is executed by the task READER_WRITER in mutual exclusion and only when it reaches an *accept* statement for the entry. The *select* statement allows the task to choose any

```
task READER_WRITER is
  procedure READ (V : out ELEM);
  entry WRITE(E : in ELEM);
end;

task body READER_WRITER is
  RESOURCE  : ELEM;
  READERS   : INTEGER := 0;

  entry START;
  entry STOP;

  procedure READ(V : out ELEM) is
    -- READ is a procedure, not an entry, hence concurrent calls of READ are possible
    -- READ synchronizes such calls with the entry calls START and STOP
  begin
    START; V := RESOURCE; STOP;
  end;

begin
  accept WRITE(E : in ELEM) do
    RESOURCE := E;
  end;

  loop
    select
      accept START;
      READERS := READERS + 1;
    or
      accept STOP;
      READERS := READERS - 1;
    or when READERS = 0 =>
        accept WRITE(E : in ELEM) do
          RESOURCE := E;
        end WRITE;
    end select;
  end loop;
end READER_WRITER;
```

FIG. 2.9. EXAMPLE TO ILLUSTRATE THE ADA MULTITASKING FACILITY

[HON 79]

one of several alternatives. In the example, the accept statements for the entries START, STOP and WRITE are the alternatives in the select statement. A condition may be associated with an alternative and acts as a 'guard' which controls when a called entry may be executed. For example, the condition READERS=0 controls the execution of the entry WRITE. A *delay* statement (not used in the given example) can be used as an alternative to allow the task to take some action in case none of the other alternatives are executable over a given period of time, either because there are no calls or because the guards do not permit execution.

Tasks are started through an *initiate* statement. If an entry or procedure is called when the task containing the procedure or entry is inactive (either because the task has not been initiated or because it has terminated) the exception TASKING_ERROR is raised in the task issuing the call.

Although communication between tasks through shared variables is possible in ADA, our program makes use only of procedure and entry calls for communication. Therefore a remote call facility has to be available to permit communication between tasks at different sites.

Apart from performance considerations, the most critical issue concerning the design of the remote call facility is that of *call semantics*. As a result of duplication of messages in the communication system or as a result of retrying a call (when the return does not come in time or because of crashes in the caller or callee) multiple executions of a procedure may occur as a result of a single call. Nelson [NEL 81] considers the alternative ways of dealing with this possibility. In *at-least-once* semantics, the results obtained by the caller may be the ones obtained from any one of multiple executions of the procedure caused by the call. In *last-of-many* semantics, the

results obtained correspond to the last of the multiple executions. Achieving the latter is complicated by the occurrence of crashes, since a crashed site can leave calls that continue to execute on other machines. Lampson [LAM 81] calls executions, whose return messages do not reach the calling task, *orphans*. In order to get last-of-many semantics, a crashed site on recovery must exterminate its orphans before retrying the call or else adopt them i.e. use their results in completing the call rather than retrying [NEL 81, LAM 81]. Neither option is inexpensive to implement.

Here we do not develop a generalized remote procedure call (RPC) mechanism but restrict ourselves to outlining a simple design that is adequate for our file update algorithm. Some simplifications arise in the case of our algorithm. First, all remote calls are to entries rather than procedures. Hence nothing has to be done to serialize the multiple executions that may be caused by a given call. Lampson's solution for last-of-many semantics given in [LAM 81] makes use of an extra unique id, which is assigned to every call and is included in all retries of the given call, to effect this serialization. Second, our algorithm does not assume extermination of orphans before the recovery of a crashed site is begun, hence one or more orphans may still be executing or awaiting execution in an entry queue when it starts its recovery. The algorithm does not require any task to retry its calls after a crash. But it is required that orphaned executions of an entry terminate before a new call to the same entry by the recovered site is started. Our requirements for the RPC are satisfied by the following design.

Each remote call is converted to a message by the RPC mechanism, timestamped and sent to the destination site. A timer is set and the message resent with a new timestamp at timer runout unless one of the following

events occurs before the runout:

(i) A return message, bearing the timestamp of the message and carrying the results of the entry execution caused by the message, is received.

(ii) The destination site is marked DOWN at the calling site. (If the site was already marked DOWN at the time the call was received from the calling task, the message is not sent at all.)

(iii) A message, bearing the timestamp of the message sent, is received, signifying that the task containing the entry is not active.

In the first case, the calling task resumes execution with the results of the return message. In the latter two cases, the exception TASKING_ERROR is raised in the calling task. The message is resent as many times as necessary until one of the above events occurs within the timeout period.

When the RPC mechanism at site i is initiated after recovering from a crash, it reads the global clock and stores the obtained value, TSTART. It ignores all received call messages bearing timestamps less than TSTART. It maintains a variable, MAX(j) for each site j which sends a remote call message to site i with a timestamp greater than TSTART. MAX(j) always carries the largest timestamp received in a call message from site j. A call message from site j is ignored unless its timestamp exceeds MAX(j).

When a call message is received by site i which is not to be ignored for either of the reasons cited above, MAX(j) is set to the timestamp of the message and

(i) if the task containing the entry is active, the call is added to the queue for the entry called. When a call completes, the results are sent in a message to the calling site with the timestamp of the call message.

(ii) if the task is not active, a message is sent to the calling site, carrying the

timestamp of the call message and signifying that the task is not active.

Our design ensures last-of-many semantics for the no-crash case. If the caller crashes, it does not, on recovery, retry any call it may have been executing at the time of the failure. Rather the task is simply restarted. Thus we do not obtain last-of-many semantics here. However, in this design all the orphaned executions of a given entry are guaranteed to terminate before the entry is executed as a result of a call made by the recovered site.

2.3.5. Interface to the Status Maintenance Mechanism

Two primitives are provided by the site maintenance scheme based on the global clock facility described in Section 2.2.

First, the function READCLOCK returns the current value of the global clock. Second, the non-blocking primitive WATCHDOWN can be invoked on any site n other than the local site. The site status maintenance software invokes a designated entry in the task invoking the watch, immediately if the site is already DOWN, otherwise when the site is next marked DOWN in the *CRASH_OTHERS* graph. The id of the site on which the watch was invoked and the current reading of the global clock are supplied as parameters when the designated entry is invoked. If WATCHDOWN is invoked by a task on a site when it already has a watch on that site which has not yet returned, the invocation has no effect.

2.3.6. Description of the Algorithm

We now give the description of the algorithm and the motivation behind the steps involved. It is assumed that the granularity of locking is the entire file. There is a version-number associated with the file which is incremented on each update. The N copies of the file are assumed to be at sites 1 through

N. The following is a description of the actions taken by each of the sites bearing a file copy in each of the states shown in Fig. 2.7.

The following actions are undertaken by site i ($1<i<N$) in state COLD:

(i) read the local clock to determine the time of entry into COLD state and store it in the variable TREC(i).

(ii) broadcast the triad (local id, local state, TREC(i)) to all other sites which have a copy of the file.

(iii) invoke WATCHDOWN on all sites which have a file copy.

(iv) send requests to all sites which have file copies to send their current state values together with the times at which they entered COLD state most recently.

(v) Wait till, for each of these sites, the required information has been obtained or the watch on it has returned.

The status information obtained is assembled into a queue (in step (vi) below) that is ordered in increasing order of the TREC values for sites which have reported their status, while the DEAD sites, from which no report was received, follow in any order after them. Consider any site j ($\neq i$) holding a file copy. If the watch on j has returned, then clearly, site j can re-enter COLD state when it recovers only with TREC(j) > TREC(i). Therefore, when it recovers and forms its queue, it will find that site i has a smaller TREC value and will put itself behind site i in its queue. Also sites which have reported larger TREC values will do the same. Thus these sites which site i will put behind it in its queue in step (vi) will put themselves behind it in theirs and thus their queues will be consistent in this sense. Similarly sites which have lower TREC values than i and which therefore are ahead of it by site i will put it after themselves in their queues.

The wait in step (v) will always terminate since for every site $j \neq i$ either

(a) site j will respond to the request in step (iv) or

(b) the WATCHDOWN placed on it at step (iii) at site i will return or

(c) if site j recovers from a crash before the WATCHDOWN on it is placed (so that the WATCHDOWN does not return) but the status request from site i in step (iv) arrives while the site has not initiated its file update software (so that the request does not get a response), then site j will send the necessary information when it itself executes step (ii).

(vi)For each site with a file copy, form the triad (site id, site state, time of entry TREC into COLD state). For sites that have had WATCHDOWN on them return, the site state is set to DEAD and the time of entry into COLD state, TREC, is set to the time supplied by the site maintenance mechanism when it returns the watch. For the others these variables are set to values obtained from their status reports. The triads are then ordered in a queue called STATUS_Q according to increasing values of TREC for the non-DEAD sites, with the triads for the DEAD sites following in arbitrary order. Moreover from now on till the site i crashes, the status of these sites is updated as status reports are received from them concerning their state transitions and as the site status maintenance mechanism reports crashes among them (a WATCHDOWN is always maintained on all file-copy-bearing sites other than i itself, which are recorded as not DEAD in the STATUS_Q.) After each update, the queue is rearranged if necessary to reflect the ordering rules mentioned above.

(vii) Initiate tasks to receive and buffer update lists broadcast by the PRIMARY.

(viii) Start a task to maintain a list of received updates so that if and when

the site becomes PRIMARY it is able to carry out its responsibility of bringing WARM sites up-to-date at regular intervals.

(ix) Wait till the first update list broadcast by the current PRIMARY arrives.

(x) Obtain the lowest version number V corresponding to an update in the first update list received.

(xi) Obtain a copy of the file warmer than V. For this purpose, a site marked WARM in the STATUS_Q is used in preference to the sites marked HOT or PRIMARY in order to avoid locking out the file to update access. The latter sites are used if no site marked WARM is in STATUS_Q.

With reference to step (viii) some explanation is required. A site *i* must, from the time it begins receiving update lists, maintain a list L of updates that it cannot be sure have reached all the appropriate sites, i.e. the sites that have initiated tasks to receive update lists in step (vii) above. Let X be the version number of the latest update it knows to have reached all the appropriate sites. (When it receives its first update list, X is set to one less than the lowest version number of an update in the list. This is because the PRIMARY always finishes sending one list of successive updates to the appropriate sites before starting the next batch.) When the next update list arrives, site *i* sets X to max (X,Y) where Y is the number one less than the lowest version number in the new list and only preserves in L those updates it has received that have version numbers greater than X. In this way it continues till it enters HOT state.

In the HOT state, site *i* itself takes part directly in every update and updates are added to L as soon as they are committed. When an update list arrives from the current PRIMARY, then, if Y has the same significance as above, all updates prior to Y are deleted, as they can be inferred to have

already received by the appropriate sites.

In the PRIMARY state, updates continue to be directly added to L as they are committed. Periodically all the updates in L are broadcast to the appropriate sites and when the broadcast is complete, the updates that have been broadcast are deleted from L.

It is possible that no update list is ever received, or even if it is, that step (xi) above cannot be completed. This can happen if there is a sequence of failures such that only WARM or COLD sites, if any, are left in front of site i in STATE_Q. In the latter case, the reason for not being able to complete step (xi) will be that these sites had not been sent the update corresponding to version number V (the lowest version number in the first update list received) when the last HOT site failed, leaving them stranded. This means not only that site i cannot enter WARM state, but that the site at the front of STATE_Q cannot enter HOT state. This is a catastrophic failure since updates can no longer be performed on the file and hence requires manual intervention to reinitialize the system. The signal for manual intervention is made by the WARM or COLD site that finds itself at the head of its STATE_Q.

After step (xi) above has been performed, the site i has entered WARM state. In this state its actions are:

(i) Broadcast the news of the state transition to the other sites having file copies.

(ii)Initiate tasks to do the following tasks:

(a) Respond to sites wishing to acquire WARM copies of the file so that they can enter WARM state.

(b) Consolidate the buffered update lists with the local copy of the file, as these lists become available.

(iii) Wait till it is time to enter HOT state.

This wait terminates when site i perceives itself to be in one of positions 1 through P in its STATE_Q. Typically this will happen when site i moves from the (P+1)th position to the Pth position as a result of a failure of one of the sites in the first P positions. But sometimes site i when entering WARM state may already find itself in one of the first P positions because of a large number of failures. When the wait terminates the following steps are executed prior to entering HOT state.

(iv) Obtain the id of the PRIMARY from the local STATE_Q.

(v) Request the PRIMARY to supply the current version number of the file. The PRIMARY locks out update access on the file for a period of time, expecting site i to complete its transition to HOT state in this time.

(vi) After getting the current version number, wait till the version number of its local copy reaches this version number as a result of the merging of received update lists.

(vii) Inform all other sites with file copies by means of a status report that it is entering HOT state.

(viii) Perform a handshake with the the site from which the version number was obtained in step (v). If the handshake is successful, this means that no updates have occurred since. Further, every site that can become PRIMARY from now on till site i itself becomes PRIMARY or crashes, has site i marked as HOT in its STATE_Q. This ensures that site i always participates in every future atomic update of HOT copies till it crashes. This guarantee holds since the update transaction co-ordinator, before committing an update, checks with the current PRIMARY to make sure that all HOT sites have signified their agreement to commit the update (see below). Hence site i can enter HOT

state. On the other hand, the handshake may not complete successfully. This may happen for one of two reasons. The site which supplied the version number in step (v) may have crashed. It may have removed the readlock it had placed on its local file copy because site i did not perform the handshake in the allotted period, in which case it will refuse to participate in the handshake. In either case, the guarantee mentioned above cannot be provided and hence site i must go back to step (iv).

It may happen that all the sites in front of site i crash before it can complete the handshake. This is again a catastrophic failure since no site can become HOT now without outside intervention. The same signaling mechanism mentioned above comes into play, i.e. site i in WARM state finds itself at the head of STATE_Q and therefore invokes manual intervention.

In HOT state, site i performs the following actions:

(i) Initiate a task to participate in performing atomic updates in co-operation with the other HOT sites, the PRIMARY and the transaction co-ordinator.
(ii) Wait till it is time to become PRIMARY.

The wait terminates when site i becomes the first in its STATE_Q. It then enters PRIMARY state, in which it performs the following actions:

(i) participate in performing atomic updates in co-operation with the HOT sites and the transaction co-ordinator.
(i) periodically broadcast all the accumulated updates, completing one broadcast before starting the next.
(ii) initiate a task to respond to requests from WARM sites for help in entering HOT state. This task does the following. When a request for the current version number is received from a WARM site, it sets a readlock on its local

file copy. It then obtains the local version number and returns it to the requesting site, at the same time initiating a broadcast of an update list carrying updates upto to the current version number, so that the requesting site can quickly make itself current. It waits for a given period of time for the requesting site to perform a handshake signifying that it has accomplished this and informed all all the sites with file copies that it is entering HOT state. If the handshake occurs within the given period, site i releases the readlock after the handshake. Otherwise the readlock is released at the end of the allotted period and site i refuses to do a handshake with the requesting site, obliging it to start all over again by asking for the current version number.

Lastly, we describe how the atomic update occurs. The co-ordinator of the update transaction sends the updates to the sites it believes to be HOT and to the site it thinks is the PRIMARY. (The site where the co-ordinator resides can, when needed obtain this information either from one of the sites with file copies, or these latter can themselves broadcast their transition into HOT and PRIMARY states to the entire network.) On receiving the update, a HOT site obtains a writelock on its local copy and responds 'ready'. The PRIMARY obtains a writelock, and then gets the set of HOT sites from its STATE_Q. If the number of HOT sites is not P-1, the PRIMARY releases the writelock and rejects the update, otherwise it responds 'ready' and sends the list of HOT sites along with its response.

The co-ordinator commits the transaction if and only if:

(i) exactly one site sends a list of HOT sites, i.e. only one site responds in PRIMARY state, along with its response.

(ii) all the sites indicated in the list and the PRIMARY respond 'ready'.

Otherwise the transaction is aborted. On receiving the commit or abort signal, the HOT sites and the PRIMARY perform or ignore the update accordingly and release the writelock.

In addition, the co-ordinator must make use of a reliable commit facility that ensures that even if the co-ordinator crashes at any time during the transaction, the sites being updated all receive an abort or all receive a commit signal. This can be done using commit backups [HAM 80]. This is to ensure that sites being updated are not left holding the writelock, not knowing whether to commit or abort the transaction.

The algorithm for the co-ordinator and its backups is given in [HAM 80] and hence is not displayed in the appendix to this chapter.

2.3.7. Choice of Parameters

The parameters of the algorithm are TBROD, the refresh interval; N, the total number of copies; and P, the number of HOT copies.

The choice of TBROD should be made on the basis of how up-to-date the information in the WARM copies is required to be, and the constraints on the buffer space in which updates being accumulated for broadcast are stored.

The value of N-P will depend on the number of sites where the frequency of read commands, which do not require the most up-to-date information, is high. It can be quite large since increasing N does not cause a penalty to be paid in the response time and immediate processing required for updates.

The value of P depends on the number of sites where the frequency of read commands which do require the latest information is high, and on the amount of protection desired against the possibility of the catastrophic failure in which no HOT copies are left with UP sites. This failure requires

manual intervention to determine which sites have the most current version and re-initialize the system.

We show below a rough computation to determine how much protection a given value of P gives against catastrophic failure.

Given the value of TBROD and the frequency and average size of updates, we can compute the amount of data that must flow from the PRIMARY to a WARM site to make its copy HOT, and thence the amount of time required. Assume that this time period is exponentially distributed with time constant $T_r = \frac{1}{\mu}$. We assume that N is large enough that there are always a sufficiently large number of sites which have been up long enough to become WARM. Thus when the set of HOT copies suffers a loss of one or more copies, the introduction of new HOT copies is not delayed by the non-availability of WARM copies.

Assume that the period for which a site is UP is exponentially distributed with a time constant $T_n = \frac{1}{\lambda}$.

We wish to find the expected time that elapses starting from a state in which P HOT copies exist to a state where none exist.

Fig 2.10 shows the state diagram with the state transitions. The updates broadcast for the purpose of a making a WARM copy HOT reach the other WARM sites in parallel, and there is always a large number of WARM sites assumed present. Therefore when a WARM site in the process of making its copy HOT fails, its place is just taken by the next WARM site in the queue and the process of making *its* copy HOT continues where it was left off for the crashed site. Therefore we need not concern ourselves with failures of WARM sites. Note that because of our assumption of exponential distributions, the

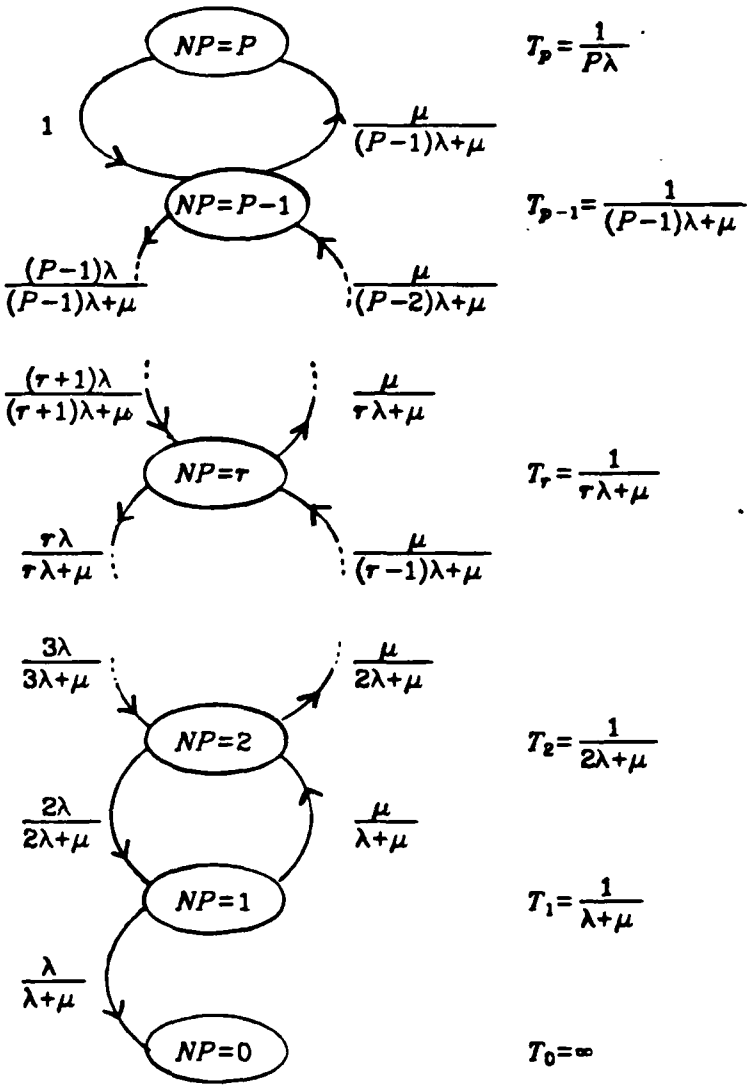


FIG. 2.10. STATE DIAGRAM SHOWING EXPECTED TIME OF STAY IN EACH STATE AND TRANSITION PROBABILITIES.

random variable $NP(t)$, the number of HOT copies at time t is memory-less.

From probability theory, it can be shown that for states $NP=r, r=1,2,...,P-1$, the time spent in the state, given the next state, has the same distribution.

$$T(r)=\frac{1}{r\lambda+\mu}$$

for $r=1,2,...P-1$, and

$$T(P)=\frac{1}{P\lambda}$$

where $T(.)$ is the mean of the time spent in a state.

Further, the transition probabilities are

$$P(r,r-1)=\frac{\lambda r}{\lambda r+\mu}$$

for $r=1,2,...P-1$,

$$P(P,P-1)=1$$

and

$$P(r,r+1)=\frac{\mu}{\lambda r+\mu}$$

for $r=1,2,3,...P-1$.

Let X_r be the mean time taken for the first transition from state $NP=r$ to state $NP=0$. Then we get

$$X_1=\frac{1}{\lambda+\mu}+\frac{\mu}{\lambda+\mu}X_2 \qquad (1)$$

$$X_r=\frac{1}{r\lambda+\mu}+\frac{\mu}{r\lambda+\mu}X_{r+1}+\frac{r\lambda}{r\lambda+\mu}X_{r-1} \qquad (2)$$

for r=2,3...P-1 and

$$X_p=\frac{1}{P\lambda}+X_{P-1} \qquad (3)$$

A closed form solution for X_p from the above equations could not be found, hence a lower bound was computed as follows.

From equation (1), we get

$$\frac{X_1}{X_2} > 1 - \frac{\lambda}{\mu}$$

Substituting in equation (2) with $r=2$, we get,

$$\frac{X_2}{X_3} > 1 - 2(\frac{\lambda}{\mu})^2$$

Proceeding in this manner we get from the substitution in equation (2) with $r=P-1$,

$$\frac{X_{p-1}}{X_p} > 1 - (P-1)!(\frac{\lambda}{\mu})^{P-1}$$

Substituting in equation (1) we get

$$X_p > \frac{1}{\lambda P!}(\frac{\mu}{\lambda})^{P-1}$$

Assuming, for instance, that a site fails once a day and that refreshing takes 5 minutes, we get, for P=2 and P=3, expected times to catastrophic failure greater than 5 months and 40 years respectively.

2.4. Conclusion

In this chapter, we proposed a status maintenance scheme for a point-to-point network. This scheme has two important features:

(a) It is based on a global clock facility. If any site i has another site j marked as *DOWN* at time t on this clock, then site j is really *DOWN* at time t. Therefore, site i can have the assurance that site j will perform no actions (visible above the status maintenance layer) from t to the time it informs site i that it is back *UP*.

(b) The marking of a site as *DOWN* is based on data gathered from its neigh-

bors, rather than its response (or lack thereof) to a probe message from an arbitrary point in the network. This prevents sites from being mistakenly marked as *DOWN*, when there is a routing failure, or the sites are heavily loaded and slow to respond, etc.

The overhead caused by the scheme is of the same order as the new Arpanet routing method. In fact some of the processing is common and can be merged.

Based on this status maintenance scheme, we developed a method for updating a replicated file. The use of the status maintenance scheme allows the sites to perform reconfiguration actions (e.g. to take over the functions of a crashed site) independently rather than making the reconfiguration decisions collectively, with all the file-copy-bearing sites taking part. The method allows read access to be performed inexpensively when it is not necessary to obtain the latest information, through the use of WARM copies. The addition of WARM copies does not cause the performance of update transactions to deteriorate.

APPENDIX

The total number of sites carrying a copy of the file is N. Further assume the total number of HOT sites plus the PRIMARY is sought to be maintained at P. Below we specify the package COMMON and a set of task families each having N members, one for each site bearing a file copy. The program for each such site consists of the package COMMON and one member of each task family.

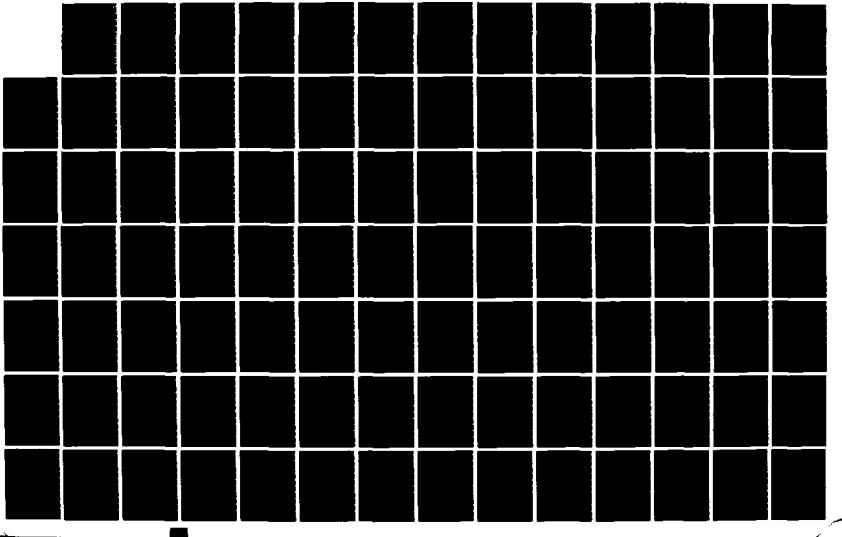
```
package COMMON is

   type SITE_STATE is (DEAD,COLD,WARM,HOT,PRIMARY);
   type SITE_ID is INTEGER range 1..N;
   type COPY_SET is array(1..N) of BOOLEAN;
   type VERSION_NUMBER is 0..SYSTEM'MAX_INT;
   type TRIAD is record
                   NO: SITE_ID;
                   STATE: SITE_STATE;
                   TR: TIME;
                 end record;--this record type is used to transmit
                            --and store site status.
   type UPDATE_PACKAGE is record
                             LVN: VERSION_NUMBER;
                             HVN: VERSION_NUMBER;
                             UPDATES: array (LVN..HVN) of UPDATE;
                       end record;--used in broadcasting update-lists
                                  --to WARM sites.
   function GET_MY_ID return TASK_ID;--returns the id of
                                     --the calling task.

end COMMON;
```

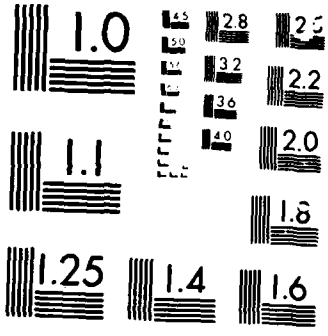
First we give the task family declarations.

```
task FILE_RECOVER(INTEGER range 1..N);

 --this task initiates the other local tasks and co-ordinates
 --the entire recovery process.


task STATUS_REPORT_SENDER(INTEGER range 1..N) is

   --this task sends the site status in response to requests.
   entry STATUS_REQUEST(NOD:SITE_ID);
```

AD-A169 247 AVAILABILITY AND CONSISTENCY OF GLOBAL INFORMATION IN COMPUTER NETWORKS(U) CALIFORNIA UNIV BERKELEY C V RAMAMOORTHY MAY 86 ARO-19159.3-EL DAAG29-83-K-0086 2/3

UNCLASSIFIED F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.8
3.2
3.6
4.0

```
end STATUS_REPORT_SENDER;


task STATUS_REPORT_RECEIVER(INTEGER range 1..N) is

  --this task receives status reports from other sites and
  --calls another task to update the
  --locally stored status information accordingly.
  entry STATUS_REPORT(T:TRIAD);

end STATUS_REPORT_RECEIVER;

task SITE_CRASH_DETECTOR(INTEGER range 1..N) is

  --this task places WATCHDOWNS on the sites bearing file copies
  --which are up and calls another task to update the locally
  --stored status information when a watch returns.
  entry SITEUP(NOD:SITE_ID);
  entry WATCH_DOWN_INTERRUPT(NOD:SITE_ID;T:TIME);

end SITE_CRASH_DETECTOR;


task UPDATE_RECEIVER(INTEGER range 1..N) is

  --this task receives update-lists from whichever site is
  --currently PRIMARY till the site in which the task resides
  --itself enters PRIMARY state.
  entry UPDATE_LIST(UP:UPDATE_PACKAGE);
  entry ENTERING_HOT;
  entry QUIT;

end UPDATE_RECEIVER;


task COPY_STATUS_KEEPER(INTEGER range 1..N) is

   --this task maintains the status of each site bearing a file
   --copy in a queue hereafter referred to as STATUS_Q.
  entry GET_PRIMARY_ID(NOD:out SITE_ID);
  entry WAIT_TO_BECOME_PRIMARY;
  entry WAIT_TO_ENTER_HOT;
  entry WAIT_TILL_INIT;
  entry GET_HOTLIST(S:out COPY_SET);
  entry GET_STATUS(NOD:SITE_ID;STATE:out SITE_STATE;T:out TIME);
  entry UPDATE(T:TRIAD);

end COPY_STATUS_KEEPER;


task UPDATE_COLLECTOR(INTEGER range 1..N) is

  --this task performs the buffering for update_lists received
```

```
  --from the PRIMARY till they are integrated into the local
  --copy of the file.
  entry ADD_TO_LIST(UP:UPDATE_PACKAGE);
  entry GET_FIRST_VERSION_NUMBER(VN:out VERSION_NUMBER);
  entry GET_FROM_LIST(UPA:out access UPDATE_PACKAGE);
  entry QUIT;

end UPDATE_COLLECTOR;


task READER_WRITER(INTEGER range 1..N) is

  --this task performs read and update operations on the local file
  --copy and provides the synchronization through locks.
  entry INITIALIZE; --creates an empty file with version number 0.
  entry GET_READLOCK(TID:TASK_ID);
    --callable for a read request from a transaction only
    --after the site has entered WARM state.
  entry GET_WRITELOCK(TID:TASK_ID);
    --as above for a write transaction but in addition the task
    --FILE_RECOVER calls this entry in procedure TRANSFER_FILE
    --to write a a WARM version into the initialized file when the
    --site holding this task is in COLD state.
  entry RELEASE_READLOCK(TID:TASK_ID);
  entry RELEASE_WRITELOCK(TID:TASK_ID);
  entry READ(...);
  entry WRITE(U:UPDATE;V:VERSION_NUMBER;FV:out VERSION_NUMBER);
     --if the current version number is 0, the update is performed
     --and the value of V is returned in FV.If the current version
     --number is not 0 and if V is not equal to one more than the
     --current version number, the procedure simply returns with
     --the current version number in FV. Else, the update is
     --performed and the version number incremented by one, and
     --the procedure returns the new version number in FV.
  entry GET_VERSION_NUMBER(V:out VERSION_NUMBER);

end READER_WRITER;


task LATEST_VERSION_NO_REQ_HANDLER(INTEGER range 1..N) is

  --this task executes when the local copy is in PRIMARY state
  --and provides the latest version number of the file to any
  --site trying to enter HOT state.
  entry HOT_VERSION_NO_REQ(NO:SITE_ID;V:out VERSION_NUMBER);
  entry HANDSHAKE(NOD:SITE_ID;V:VERSION_NUMBER;
                  ST:out (SUCCESS,FAILURE));

end LATEST_VERSION_NO_REQ_HANDLER;


task UPDATES_CONSOLIDATOR(INTEGER range 1..N) is
```

```
  --this task does the merging of buffered update_lists
  --into the local file copy when in WARM state.
  procedure WAKE_AT(V:VERSION_NUMBER);
  entry WAKEUP;
  entry QUIT;

end UPDATES_CONSOLIDATOR;


task UPDATE_HANDLER(INTEGER range 1..N) is

  --this task along with peer tasks in the PRIMARY and
  --HOT sites and the transaction coordinator atomically
  --updates the HOT and PRIMARY copies.
  entry UPDATE(U:UPDATE);
  entry BECOMING_PRIMARY;

end UPDATE_HANDLER;


task UPDATE_LIST_BROADCASTER(INTEGER range 1..N) is

  --this task does the broadcasting of update-lists
  --in PRIMARY state.
  entry PICKUP_PACKAGE(UP:UPDATE_PACKAGE);

end UPDATE_LIST_BROADCASTER;


task BROADCAST_TIMER(INTEGER range 1..N);
  --this task designates the times for the periodic update-list
  --broadcasts.


task UPDATES_TO_BE_BROADCAST_MAINTAINER(INTEGER range 1..N) is

  --from the time that the site holding this task starts receiving
  --update-lists, this task maintains a list of updates that it is
  --not certain have been received by all appropriate sites; it
  --provides the update-lists to be broadcast in PRIMARY state.
  entry REPLACE(UP:UPDATE_PACKAGE);
  entry ENTERING_HOT;
  entry DELETE_UPTO(VN:VERSION_NUMBER);
  entry PREPARE_PACKAGE;
  entry ADD_UPD(U:UPDATE;V:VERSION_NUMBER);
  entry BROADCAST_ON_REACHING(V:VERSION_NUMBER);

end UPDATES_TO_BE_BROADCAST_MAINTAINER;
```

Next we specify the task bodies.

```
task body FILE_RECOVER is

   LNI: constant := FILE_RECOVER'INDEX;
   TREC: constant TIME;
   LOCAL_STATE: SITE_STATE;

   procedure BROADCAST_STATE(S:SITE_STATE) is
                  --used to broadcast state transitions;
   begin
     for I in SITE_ID loop
        if LNI ≠ I then
         begin
            STATUS_REPORT_RECEIVER(I).STATUS_REPORT((LNI,S,TREC));
         exception
            when TASKING_ERROR=>;--ignore exception if call
                                 --does not complete
         end;
        end if;
     end loop;
   end BROADCAST_STATE;

   procedure FILE_TRANSFER (V:VERSION_NUMBER) is
    --This procedure obtains a copy of the file 'warmer' than V.
    --It uses a WARM site in preference to a HOT or the
    --PRIMARY site to avoid interference in updating them.
    --It will not return if all the sites which received
    --the update corresponding to version number V and ahead
    --of the caller in STATE_Q fail before the transfer completes.
    --In this case, some site not in HOT or PRIMARY state will find
    --itself at the head of STATE_Q and invoke manual intervention.
      .
      .
      .
   end FILE_TRANSFER;


begin

   initiate COPY_STATUS_KEEPER(LNI);
   initiate STATUS_REPORT_RECEIVER(LNI);
   initiate READER_WRITER(LNI);

   LOCAL_STATE:=COLD;
   TREC:=READCLOCK;

   COPY_STATUS_KEEPER(LNI).UPDATE((LNI,LOCAL_STATE,TREC));
                --initialize the queue element for the site
                --in which the task resides, in STATUS_Q.
   initiate STATUS_REPORT_SENDER(LNI);
                   --after the above initialization,status
                   --requests can be replied to.
   BROADCAST_STATE(COLD); --inform all file-copy-bearing sites of
                          --local status.
```

```
initiate SITE_CRASH_DETECTOR(LNI);--this task initially places
                                  --WATCHDOWNs on all other
                                  --file-copy-bearing sites.
for NOD in SITE_ID loop
  if NOD ≠ LNI then
     STATUS_REPORT_SENDER(NOD).STATUS_REQUEST(LNI);
  end if;              --to all other file-copy-bearing sites
end loop;              --send status requests.
COPY_STATUS_KEEPER(LNI).WAIT_TILL_INIT;
    --at this point the state of all sites is initialized in
    --STATE_Q; either WATCHDOWN on them has returned or they
    --have returned status reports.

declare--in this block a WARM copy is obtained.
  INV:VERSION_NUMBER;
begin
  initiate UPDATE_COLLECTOR(LNI);
  initiate UPDATES_TO_BE_BROADCAST_MAINTAINER(LNI);
  initiate UPDATE_RECEIVER(LNI);
  UPDATE_COLLECTOR(LNI).GET_FIRST_VERSION_NUMBER(INV);
  FILE_TRANSFER(INV);
end;

LOCAL_STATE:=WARM;
BROADCAST_STATE(WARM);--inform all other sites of
                      --the state transition;
COPY_STATUS_KEEPER(LNI).UPDATE((LNI,LOCAL_STATE,TREC));
                      --update STATE_Q;
initiate UPDATES_CONSOLIDATOR(LNI);
COPY_STATUS_KEEPER(LNI).WAIT_TO_ENTER_HOT;
                --wait terminates when the
                --site holding the task enters one of the
                --first P positions in STATE_Q.

declare --in this block the site makes its file copy
        --correspond with the latest version.
  P:SITE_ID;
  BDONE: BOOLEAN:=FALSE;
  DONE,RES1,RES2:(SUCCESS,FAILURE) :=FAILURE;
  HOT_VN:VERSION_NUMBER;
begin
  while RES1=FAILURE loop
              -- loop till successful handshake occurs.
     while RES2=FAILURE loop --loop till latest version
                             -- number is obtained.
        begin
          COPY_STATUS_KEEPER(LNI).GET_PRIMARY_ID(P);
          LATEST_VERSION_NO_REQ_HANDLER(P).
                        HOT_VERSION_NO_REQ(LNI,HOT_VN);
          RES2:=SUCCESS;
        exception
          when TASKING_ERROR =>; --continue inner loop.
        end;
```

```
            end loop;
            UPDATES_CONSOLIDATOR(LNI).WAKE_AT(HOT_VN);
                      --wait terminates when the version number reaches
                      --HOT_VN.
            if not BDONE then BROADCAST_STATE(HOT); BDONE:=TRUE; end if;
                      --inform sites ahead in STATE_Q that the site is
                      --entering HOT state if this has not been done in
                      --previous iterations of the loop.
            begin
             LATEST_VERSION_NO_REQ_HANDLER(P).
                           HANDSHAKE(LNI,HOT_VN,DONE);
             if DONE=SUCCESS then
                RES1:=SUCCESS;  --handshake terminates successfully.
             else RES1:=FAILURE;--handshake fails.
             end if;
            exception
              when TASKING_ERROR =>
                           --the primary has failed since
                           --it supplied the latest version number.
                RES1:=FAILURE;
            end;
          end loop;
          UPDATES_COLLECTOR(LNI).QUIT;
          UPDATES_CONSOLIDATOR(LNI).QUIT;
          UPDATE_RECEIVER(LNI).ENTERING_HOT;
          UPDATES_TO_BE_BROADCAST_MAINTAINER(LNI).ENTERING_HOT;
        end;

        LOCAL_STATE:=HOT;
        COPY_STATUS_KEEPER(LNI).
                     UPDATE((LNI,LOCAL_STATE,TREC));--update STATE_Q.
        initiate UPDATE_HANDLER(LNI);
        COPY_STATUS_KEEPER(LNI).WAIT_TO_BECOME_PRIMARY;

        UPDATE_RECEIVER(LNI).QUIT;
        UPDATE_HANDLER(LNI).BECOMING_PRIMARY,
        initiate LATEST_VERSION_NO_REQ_HANDLER(LNI);
             --respond to requests for the latest version number
             --from sites wanting to enter HOT state.
        initiate UPDATE_LIST_BROADCASTER(LNI);
           --commence periodic broadcasts of update lists
        initiate BROADCAST_TIMER(LNI);
        LOCAL_STATE:=PRIMARY;
        COPY_STATUS_KEEPER(LNI).
                UPDATE((LNI,LOCAL_STATE,TREC));--update STATE_Q.
        BROADCAST_STATE(PRIMARY);--inform all other file-copy bearing
                                 --sites of state transition.
    end FILE_RECOVER;


    task body STATUS_REPORT_SENDER is
      LNI:constant SITE_ID:=STATUS_REPORT_SENDER'INDEX;
      S:SITE_STATE;
```

```
  T:TIME;

begin
  loop
     accept STATUS_REQUEST(NOD:SITE_ID);
     COPY_STATUS_KEEPER(LNI).GET_STATUS(LNI,S,T);
     begin
       STATUS_REPORT_RECEIVER(NOD).STATUS_REPORT((LNI,S,T));
     exception
       when TASKING_ERROR =>;
     end;
   end loop;

end STATUS_REPORT_SENDER;


task body STATUS_REPORT_RECEIVER is

begin
  loop
    accept STATUS_REPORT(T:TRIAD) do
      if (T.STATE=COLD) then
           SITE_CRASH_DETECTOR(LNI).SITEUP(T.NO);
                --T.NO is back up, so a WATCHDOWN should
                --be placed on it.
      end if;
      COPY_STATUS_KEEPER(LNI).UPDATE(T);--update STATE_Q.
    end STATUS_REPORT;
   end loop;

end STATUS_REPORT_RECEIVER;


task body SITE_CRASH_DETECTOR is

LNI:constant SITE_ID := SITE_CRASH_DETECTOR'INDEX;
begin
  for NOD in SITE_ID loop --place WATCHDOWN on file copy
                          --bearing sites.
    if (NOD ≠ LNI) then
       WATCHDOWN(NOD);
    end if;
  end loop;
  loop
     select
        accept WATCH_DOWN_INTERRUPT(NOD:SITE_ID;T:TIME);
          --this entry is invoked by the status
          --maintenance scheme when a watch returns.
        COPY_STATUS_KEEPER(LNI).UPDATE((LNI,DEAD,T));
     or
        accept SITEUP(NOD:SITE_ID);
        WATCHDOWN(NOD);
     end select;
```

```
  end loop;

end SITE_CRASH_DETECTOR;


task body UPDATE_RECEIVER is

  HOT:BOOLEAN:=FALSE;
  LNI:constant SITE_ID := UPDATES_RECEIVER'INDEX;
begin
  loop
    select
     when not HOT=>
      accept UPDATE_LIST(UP:UPDATE_PACKAGE);
      UPDATES_TO_BE_BROADCAST_MAINTAINER(LNI).REPLACE(UP);
             --receipt of this list may modify the updates that
             --are known to have been broadcast.
      UPDATE_COLLECTOR(LNI).ADD_TO_LIST(UP);
                --add this list to buffered update-lists.
    or
      accept ENTERING_HOT;
      HOT:=TRUE;
    or
      when HOT=>
      accept UPDATE_LIST(UP:UPDATE_PACKAGE);
      UPDATES_TO_BE_BROADCAST_MAINTAINER(LNI).
                                DELETE_UPTO(UP.LVN-1);
              --updates upto version number UP.LVN-1 must have
              --already been broadcast.
    or
      accept QUIT;--PRIMARY state is being entered, from now on
                  --the site holding this task will do the
                  --broadcasting of update-lists.
      exit;
    end select;
  end loop;

end UPDATES_RECEIVER;


task body UPDATE_COLLECTOR is

  LNI:constant SITE_ID:=UPDATE_COLLECTOR'INDEX;
  type LIST_ELEM is record
                      LUP:UPDATE_PACKAGE;
                      SUCC:access LIST_ELEM;
                    end record;--buffer for update-list.
  HP,TP:access LIST_ELEM;
  MIN_VN,MAX_VN:VERSION_NUMBER;
                  --MIN_VN is the first update received after
                  --entry into COLD-state;MAX_VN is the latest
                  --update received.
  WAITING:BOOLEAN:=FALSE;--signifies when TRUE that the task
```

```
                              --UPDATES_CONSOLIDATOR is waiting for
                              --the next update list.
begin
  accept ADD_TO_LIST(UP:UPDATE_PACKAGE);
  HP:=TP:=new LIST_ELEM(LUP=>UP,SUCC=>null);
  MIN_VN:=UP.LVN;MAX_VN:=UP.HVN;
  loop
    select
      accept GET_FIRST_VERSION_NUMBER(VN:out VERSION_NUMBER) do
      VN:=MIN_VN;
      end GET_FIRST_VERSION_NUMBER;
    or
      accept ADD_TO_LIST(UP:UPDATE_PACKAGE);
      if (UP.HVN>MAX_VN) then--if the new list contains updates
                              --not already received.
         if (TP:=null) then--if all received updates have already
                    --been picked up by UPDATES_CONSOLIDATOR.
            HP:=TP:=new LIST_ELEM(LUP=>UP,SUCC=>null);
         else
            TP.SUCC:=new LIST_ELEM(LUP=>UP,SUCC=>null);
            TP:=TP.SUCC;
         end if;
         MAX_VN:=UP.HVN;
         if WAITING then
            UPDATES_CONSOLIDATOR(LN1).WAKEUP;
            WAITING:=TRUE;--one WAKEUP for each time GET_FROM_LIST
                          --returns zero updates.
         end if;
      end if;
    or
       accept GET_FROM_LIST(UPA:out access UPDATE_PACKAGE) do
        if (HP=null) then--if no updates on hand
         UPA:=new UPDATE_PACKAGE(lvn=>0,hvn=>0);
                        --return a null list.
         WAITING:=TRUE;--remember that UPDATES_CONSOLIDATOR will
                        --be waiting to be informed when some
                        --updates are available.
        else
         UPA:= new UPDATE_PACKAGE(HP.all);
         HP:=HP.all;
        end if;
       end GET_FROM_LIST;
    or
       accept QUIT;exit;--the site holding this task is entering
                        --HOT state.
    end select;
  end loop;

end UPDATE_COLLECTOR;


task body UPDATES_CONSOLIDATOR is
```

```
TID:constant TASK_ID:=COMMON.GET_MY_ID;
WAITING:=BOOLEAN:=FALSE;--when TRUE, this variable signifies that
            --this task is waiting for more updates to arrive.
P: access UPDATE_PACKAGE;
CVN:VERSION_NUMBER;--current version number of the local copy.
TRAP_VN:VERSION_NUMBER;
TRAP_SET:BOOLEAN:=FALSE;
LN1:constant SITE_ID:=UPDATES_CONSOLIDATOR'INDEX;

entry SET_TRAP(V:VERSION_NUMBER);
entry REACHED;

procedure WAKE_AT(V:VERSION_NUMBER) is
  --is used by FILE_RECOVER to be informed when the local copy
  --reaches the latest version number, so that it can enter
  --HOT state.
begin
  SET_TRAP(V);
  REACHED;--the latest version number has been reached.
end;
begin
  READER_WRITER(LN1).GET_VERSION_NUMBER(CVN);
  <<OUTER>>
  loop
    while not WAITING loop--loop till all update lists received
                          --have been merged into the local copy.
      UPDATE_COLLECTOR(LN1).GET_FROM_LIST(P);
      if (P.HVN=0) then--if a null list has been obtained
        WAITING:=TRUE;--then wait till some updates arrive.
      else
        READER_WRITER(LN1).GET_WRITELOCK(TID);
        for J in P.LVN..P.HVN loop--merge updates.
          if J=CVN+1 then
            READER_WRITER(LN1).WRITE(P.UPDATES(J),J,CVN);
          end if;
        end loop;
        READER_WRITER(LN1).RELEASE_WRITELOCK(TID);
      end if;
    end loop;
    <<INNER>>
    loop
      select
         accept SET_TRAP(V:VERSION_NUMBER);
         TRAP_VN:=V;
         TRAP_SET:=TRUE;
      or
       when (TRAP_SET and then CVN ≥ TRAP_VN)=>
          --version number has reached the latest version number;
         accept REACHED;
      or
         accept WAKEUP;WAITING:=FALSE;exit INNER;
          --go to process the newly arrived updates.
      or
```

```
            accept QUIT; exit OUTER;
             --site is entering HOT state.
          end select;
        end loop;
      end loop;

end UPDATES_CONSOLIDATOR;


task body COPY_STATUS_KEEPER is
 type SITE_REC is
      record
        T:TRIAD;
        PRED,SUCC:access SITE_REC;
      end record;
 HOTLIST:COPY_SET:=(1..N=>FALSE);
 SITES_NOT_INITIALIZED:=(1..N=>TRUE);
 INIT:BOOLEAN:=FALSE;
 ENTER_HOT:BOOLEAN:=FALSE;
 BECOME_PRIMARY:BOOLEAN:=FALSE;
 PRIMARY_KNOWN:BOOLEAN:=FALSE;
 HP,TP:access SITE_REC:=null;--head and tail pointers to STATE_Q.
 LOCAL_POS:INTEGER range 1..N;--position of site in which task
                              --resides in STATE_Q.
 LNI:constant SITE_ID:=COPY_STATUS_KEEPER'INDEX;

 procedure RETRIEVE (N:SITE_ID;S:out SITE_STATE;T:out TIME) is
   --This procedure searches the queue to find the element for
   --the site corresponding to N and returns the values of
   --T.STATE and T.TR.
   .
   .
 end RETRIEVE;

 procedure MODIFY(T:TRIAD) is
   --This procedure changes the queue-element for the site
   --specified in T to the values specified in T provided T.TR is
   --greater than or equal to the corresponding component of
   --the queue element; it then moves the element if necessary
   --so that STATE_Q is still sorted in increasing order of
   --the value of this component for non-DEAD sites with the DEAD
   --sites following in any order.
   .
   .
   .
 end MODIFY_REC;

 procedure GET_LOCAL_POS(POS:out INTEGER) is
   --This procedure gets the position of the queue-element
   --corresponding to site LNI, measured from the head of the
   --queue,i.e.if site LNI were at the head, the procedure will
   --return with POS=1.
   .
```

```
 end GET_LOCAL_POS;

begin
 for NOD in SITE_ID loop --form STATE_Q.
   if (TP=null) then
     HP:=TP:=
      new SITE_REC(triad=>(NOD,DEAD,0),PRED=>null,SUCC=>null);
   else
     TP.SUCC:=
      new SITE_REC(triad=>(NOD,DEAD,0),PRED=>TP,SUCC=>null);
     TP:=TP.SUCC;
 end loop;
 LOCAL_POS:=LNI;
 loop
   select
     when PRIMARY_KNOWN =>
       accept GET_PRIMARY_ID(NOD:out SITE_ID) do
        NOD:=HP.T.NO;--PRIMARY is at the head of STATE_Q.
       end GET_PRIMARY_ID;
     when BECOME_PRIMARY=>
       accept WAIT_TO_BECOME_PRIMARY;
     when ENTER_HOT=>
       accept WAIT_TO_ENTER_HOT;
     when INIT=>
       accept WAIT_TILL_INIT;
   or
     when INIT=>
       accept GET_HOTLIST(S:out COPY_SET) do
        S:=HOTLIST;
       end GET_HOTLIST;
   or
       accept GET_STATUS(NOD:SITE_ID;STATE:out SITE_STATE;
                                          T:out TIME) do
        RETRIEVE(NOD,STATE,T);
       end GET_STATUS;
   or
       accept UPDATE(T:TRIAD);
       MODIFY(T);
       if (T.STATE=HOT) or (T.STATE=PRIMARY) then
         HOTLIST(T.NO):=TRUE;
       end if;
       if not INIT then
         SITES_NOT_INITIALIZED(T.NO):=FALSE;
         if SITES_NOT_INITIALIZED=(1..N=>FALSE) then
           INIT:=TRUE;
         end if;
       end if;
       GET_LOCAL_POS(LOCAL_POS);
       if (LOCAL_POS ≤ P) then ENTER_HOT:=TRUE;end if;
       if (LOCAL_POS=1) then
          BECOME_PRIMARY:=TRUE;
       if (HP.T.STATE=PRIMARY) then
```

```
            PRIMARY_KNOWN:=TRUE;
          else
            PRIMARY_KNOWN:=FALSE;
          end if;
          if INIT and ((LOCAL_POS=1) and ((HP.T.STATE ≠ HOT)
                    or (HP.T.STATE ≠ PRIMARY)) then
           {signal and wait for manual intervention to reinitialize
            the file copy bearing sites };
          end if;--the site at the head of STATE_Q is unable to enter
                 --HOT state, therefore no further update
                 --transactions can be processed.
     end select;
   end loop;

end COPY_STATUS_KEEPER;


task body UPDATE_HANDLER is
  LNI:constant SITE_ID:=UPDATE_HANDLER'INDEX;
  HOT:BOOLEAN:=TRUE;--flag to distinguish whether the state
                    --is HOT or PRIMARY.
  CVN:VERSION_NUMBER;--current version number.
  TID:constant TASK_ID:=COMMON.GET_MY_ID;
  TR_ID: TRANSACTION_ID;
  COUNT: INTEGER range 1..N;
  READY:BOOLEAN;
  UP:UPDATE;
begin
  READER_WRITER(LNI).GET_VERSION_NUMBER(CVN);
  <<MAIN>>
  loop
    select
      accept UPDATE(U_ID:TRANSACTION_ID;U:UPDATE;
                    AM_PRIMARY:out BOOLEAN;RES:out (ACCEPT,REJECT);
                    HL:out COPY_SET) do
       --protection and integrity checks are assumed
       --to have been done.
       TR_ID:=U_ID;
       UP:=U;
       READER_WRITER(LNI).GET_WRITELOCK(TID);
       if HOT then --in HOT state
         AM_PRIMARY:=FALSE;
         RES:=ACCEPT;
         READY:=TRUE;
       else --in PRIMARY state
         AM_PRIMARY:=TRUE;
         COPY_STATUS_KEEPER(LNI).GET_HOTLIST(HL);
         for I in SITE_ID loop
          if HL(I)=TRUE then
            COUNT:=COUNT+1;
          endif;
         end loop;
         if COUNT ≠ P then --accept update only if P HOT copies
```

```
                                    --(including the PRIMARY) exist.
                READY:=FALSE;
                RES:=REJECT;
                READER_WRITER(LNI).RELEASE_WRITELOCK(TID);
              else
                READY:=TRUE;
                RES:=ACCEPT;
              end if;
            end if;
          end UPDATE;
          <<COMMIT_OR_ABORT>>
          if READY then
            loop
              accept DECISION (U_ID:TRANSACTION_ID;COMMIT:BOOLEAN) do
                  if TR_ID=U_ID then
                     if COMMIT then
                       READER_WRITER(LNI).WRITE(UP,CVN+1,CVN);
                       UPDATES_TO_BE_BROADCAST_MAINTAINER(LNI).
                                        ADD_UPD(UP,CVN);
                     end if;
                     READER_WRITER(LNI).RELEASE_WRITELOCK(TID);
                     exit COMMIT_OR__ABORT;
                        --exit this loop only if the fate of the
                        --transaction has been decided.
                  end if;
              end DECISION;
            end loop;
          end if;
        or
          accept BECOMING_PRIMARY;
          HOT:=FALSE;
        end select;
      end loop;

end UPDATE_HANDLER;


task body LATEST_VERSION_NO_REQ_HANDLER is

  LNI:constant SITE_ID:=LATEST_VERSION_REQ_HANDLER'INDEX;
  NOD:SITE_ID;
  ST:(SUCCESS,FAILURE);
  VN:VERSION_NUMBER;
  TID:TASK_ID:=GET_MY_ID;
  T: constant TIME:=..;--should be set to a value sufficient for
                       --the site requesting the latest version
                       --number to get all updates up to this
                       --version number, inform all sites of its
                       --entry into HOT state and then perform a
                       --a handshake with this task.
begin
  loop
    loop
```

```
        select
          accept HOT_VERSION_NO_REQ(NO:SITE_ID;V:out VERSION_NUMBER)
                                                                  do
           NOD:=NO;
           READER_WRITER(LNI).GET_READLOCK(TID);
            --block updates while the process of getting the
            --caller site into HOT state is going on.
              READER_WRITER(LNI).GET_VERSION_NUMBER(VN);
           V:=VN;
          end HOT_VERSION_NO_REQ;
          UPDATES_TO_BE_BROADCAST_MAINTAINER(LNI).
                    BROADCAST_ON_REACHING(VN);
            --initiate a quick update list broadcast so that
            --the caller site does not have to wait till the
            --next of the periodic broadcasts.
          exit; --go to wait for handshake;
        or
          accept HANDSHAKE (NO:SITE_ID;V:VERSION_NUMBER;ST:out
                    (SUCCESS,FAILURE)) do
            ST:=FAILURE;--a HANDSHAKE entry accepted here indicates
                        --that the call did not come in time.
          end HANDSHAKE;
        end select;
      end loop;

      loop
        select
           accept HANDSHAKE(NO:SITE_ID;V:VERSION_NUMBER;ST:out
                                              (SUCCESS,FAILURE)) do
            if (NO=NOD) and (V=VN) then
              ST:=SUCCESS; --succesful broadcast
              READER_WRITER(LNI).RELEASE_WRITELOCK(TID);
              exit; --go to wait for the next request for the latest
                    --version number.
            else
              ST:=FAILURE; --this call did not come within the set
                           --period or is a duplicate of a call that
                           --either did not come in time or which
                           --resulted in a successful handshake.
            end if;
           end HANDSHAKE;
        or
           delay T;  --time period for the site that requested the
                     --latest version number to call HANDSHAKE.
           READER_WRITER(LNI).RELEASE_WRITELOCK(TID);
           exit;
             --expected handshake did not occur, so go back to wait
             --for the next request for the latest version number.
        end select;
      end loop;
    end loop;

  end UPDATE_HANDLER;
```

```
task body UPDATE_LIST_BROADCASTER is

  LNI:constant SITE_ID:=UPDATE_LIST_BROADCASTER'INDEX;
begin
  loop
    accept PICKUP_PACKAGE(UP:UPDATE_PACKAGE);
    for all NOD in SITE_ID loop
      if LNI ≠ NOD then
        begin
          UPDATE_LIST_RECEIVER(NOD).UPDATE_LIST(UP)
        exception
          when TASKING_ERROR=>;
        end;
      end if;
  end loop;

end UPDATE_LIST_BROADCASTER;


task body BROADCAST_TIMER is

  LNI:constant SITE_ID:=BROADCAST_TIMER'INDEX;
  TBROD:constant TIME:=....; --period of update lists broadcasts.
begin loop
     delay TBROD;
     UPDATES_TO_BE_BROADCAST_MAINTAINER(LNI).PREPARE_PACKAGE;
  end loop;

end BROADCAST_TIMER;


task body UPDATES_TO_BE_BROADCAST_MAINTAINER is
 LNI:constant SITE_ID:=UPDATES_TO_BROADCAST_MAINTAINER'INDEX;
 LUPV,V:VERSION_NUMBER;  --LUPV is the version number up to which
                         --updates have already been broadcast.
 P,Q:access UPDATE_PACKAGE;
 LO,HI:VERSION_NUMBER;
 HOT:BOOLEAN:=FALSE;
 TRAP_SET:BOOLEAN:=FALSE; --used to initiate special broadcasts
                          --when a site entering HOT state.
 TRAP_VN_NO:VERSION_NUMBER;
 NO_UPD_ON_HAND:BOOLEAN:=TRUE;
begin
 UPDATE_COLLECTOR(LNI).GET_FIRST_VERSION_NUMBER(V);
 LUPV:=V-1;
 loop
   select
      when not HOT=>
         accept REPLACE(UP:UPDATE_PACKAGE);
         if NO_UPD_ON_HAND then --if no updates are in this
                                --site's possession which have
                                --not been broadcast
            if (UP.HVN>LUPV) then --if this list contains any
```

```
                                    --updates not known to have
                                    --been broadcast
                NO_UPD_ON_HAND:=FALSE;
                P:=new UPDATE_PACKAGE(LVN=>LUPV+1,HVN=>UP.HVN);
                for VN in P.LVN..P.HVN loop  --store the list.
                     P.UPDATES(VN):=UP.UPDATES(VN);
                end loop;
             end if;
          else --this site has some updates that it cannot be sure
               --have been broadcast
             if (UP.HVN>P.HVN) then --if this list contains some
                                    --new updates
               Q:=new UPDATE_PACKAGE(P.all);
               if (P.LVN>UP.LVN) then
                 LO:=P.LVN;
               else
                 LO:=UP.LVN;
               end if;   --updates with version number less than
                         --LO are known to have been broadcast.
               HI:=UP.HVN;
               LUPV:=LO-1;
               P:=new UPDATE_PACKAGE(LVN=>LO,HVN=>HI);
               for V in LO..HI loop
                  if V in Q.LVN..Q.HVN then
                    P.UPDATES(V):=Q.UPDATES(V);
                  else
                    P.UPDATES(V):=UP.UPDATES(V);
                  end if;
               end loop; --merge the newly arrived list with the
                         --updates on hand.
             end if;
          end if;
or
   when not HOT=>
      accept ENTERING_HOT;
      HOT:=TRUE; --from now on updates are directly added to
                 --the set on hand as they are committed,
                 --instead of being received in periodic
                 --broadcasts.
or
   when HOT and not TRAP_SET=>
      accept BROADCAST_ON_REACHING(V:VERSION_NUMBER);
      if (NO_UPD_ON_HAND and LUPV < V) or
          ((not NO_UPD_ON_HAND) and P.HVN < V) then
            --do not set trap if the update corresponding to V
            --has already been broadcast or if not is in the
            --the set of updates on hand.
         TRAP_SET:=TRUE;
         TRAP_VN_NO:=V;
      else --if the update has not been broadcast but is at
           --hand then initiate a broadcast.
         if ((not NO_UPD_ON_HAND) and
                 P.LVN ≤ V and P.HVN ≥ V ) then
```

```
                UPDATE_LIST_BROADCASTER(LN1).PICKUP_PACKAGE(P.all);
                NO_UPD_ON_HAND:=TRUE;
                LUPV:=P.HVN;
              end if;
          end if;
      or
          when HOT=>
              accept ADD_UPD(U:UPDATE;V:VERSION_NUMBER);
              if (NO_UPD_ON_HAND and V=LUPV+1) then
                  P:=new UPDATE_PACKAGE(LVN=>V,HVN=>V,UPDATES(V)=>U);
                  NO_UPD_ON_HAND:=FALSE;
              elseif ((not NO_UPD_ON_HAND) and V=P.HVN+1) then
                  Q:=new UPDATE_PACKAGE(P.all);
                  P:=new UPDATE_PACKAGE(LVN=>P.LVN,HVN=>V);
                  for VN in P.LVN..P.HVN loop
                    P.UPDATES(VN):=Q.UPDATES(VN);
                  end loop;
                  P.UPDATES(V):=U;
              end if;
              if (TRAP_SET and then P.HVN ≥ TRAP_VN_NO) then
                  --if the trap is set and the new update sets it off
                 UPDATE_LIST_BROADCASTER(LN1).PICKUP_PACKAGE(P.all);
                     --initiate a broadcast.
                 NO_UPD_ON_HAND:=TRUE;
                 LUPV:=P.HVN;
                 TRAP_SET:=FALSE;
              end if;
      or
          when HOT=>
              accept PREPARE_PACKAGE; --periodic broadcast
              if (not NO_UPD_ON_HAND) then
                UPDATE_LIST_BROADCASTER(LN1).PICKUP_PACKAGE(P.all);
                NO_UPD_ON_HAND:=TRUE;
                LUPV:=P.HVN; end if;
      or
          when HOT=>
              accept DELETE_UPTO(V:VERSION_NUMBER);
              if (not NO_UPD_ON_HAND) then
                if (P.HVN ≤ V) then
                    LUPV:=V;
                    NO_UPD_ON_HAND:=TRUE;
                elseif (P.LVN ≤ V) then
                    Q:=new UPDATE_PACKAGE(P.all);
                    P:=new UPDATE_PACKAGE(LVN=>V+1,HVN=>Q.HVN);
                    for VN in P.LVN..P.HVN loop
                       P.UPDATES(VN):=Q.UPDATES(VN);
                    end loop;
                end if;
              end if;
      end select;
    end loop;
  end UPDATES_TO_BE_BROADCAST_MAINTAINER;
```

CHAPTER 3

ENSURING THE CORRECTNESS OF GLOBAL INFORMATION

3.1. Introduction

In this chapter, we address the problem of maintaining the availability of global information in a computer network, in the presence of malfunctioning sites in the network.

Our model of the network is that it consists of a set of sites attached to a communication subsystem. We assume that this subsystem provides perfect *site-to-site* communication so that all messages are delivered intact in a known period. Note that in this model, the communication subsystem does not provide a reliable *broadcast* mechanism and in fact the difficulty of performing a reliable broadcast will be a major issue in the following discussion. Further it is assumed that no site A can masquerade as another site B and send messages as originating from B. The ideas presented below can be extended to the case where the communication is imperfect; the assumption of perfection is made to simplify the presentation.

In Chapter 1, we distinguished between two kinds of failure models for network sites. In the model in which *crashes* are the only mode of failure, a site exhibits *fail-stop* behavior [SCH 83] and performs a recovery procedure as its first act after each crash. In the other model, *malfunctions* are the mode of failure. A malfunctioning site may go through arbitrary state transformations and emit arbitrary messages. In the extreme case, a malfunctioning site may exhibit malicious intelligence attempting to disrupt the functioning of the rest of the network. In Chapter 2, we showed how to

detect site crashes and maintain a view of network status for the former model. The essence of malfunction as a model of failure, however, is that the existence of a malfunctioning site may go undetected for an indefinite period of time. Hence, it is necessary to develop techniques that preserve the availability of global information in the presence of arbitrary, undetected failures, and this is the aim of this chapter.

In Section 3.2., we discuss why and under what conditions replication should be used to deal with malfunctions. Section 3.3. explains the phenomenon of error propagation which can occur when malfunctioning sites are present, and which can progressively render all the global information stored in the network incorrect. Section 3.4. outlines our approach to preserving correctness. Section 3.5. deals with relevant past work in this area and the reasons why it cannot be directly used to solve the problem being considered. Section 3.6. describes the extensions to, and modifications of this work necessary to carry out our approach. Section 3.7. further develops our approach by describing protocols that prevent error propagation when a particular form of bound on the number of malfunctioning sites holds, and which have some nice properties.

3.2. Redundancy Techniques for Storing Information

In order to protect the availability of information against crashes and malfunctions, some form of redundancy is required. One form of redundancy is *replication*, in which multiple copies of the information are made and stored with each copy at a different site. Another form of redundancy that could be used is *error-detecting* and *correcting codes*. Consider a n-bit piece of information. It could be encoded in $N=n+k$ bits, where $k=\lceil \log_2 N \rceil$ using a distance-3 Hamming code and stored in N sites, one bit in each site. Then

the piece of information would remain available, as long as no more than one site crashes or malfunctions. However this solution will incur a large amount of communication overhead, since a large number of sites may have to be consulted to retrieve the information. Also since the information is partitioned among many sites, it is not possible to process it locally at any of these sites. Rather, the information must be first assembled at some point before processing, further increasing the communication overhead. Since communication bandwidth is, and is expected to remain [OUS 80], a bottleneck in most distributed systems, we do not consider this approach further. Thus although error-detecting and correcting codes can be used locally at each site e.g. in the memory and ALU, to lessen the likelihood of its crashing or malfunctioning, and also in the communication subsystem, the appropriate redundancy technique for stored information at the system level where the unit of failure is a site, is to replicate the information at multiple sites.

In order to preserve the availability of an item of information in the presence of m malfunctioning sites, it is necessary to replicate the information at $2m+1$ sites. Then by consulting each of the $2m+1$ replicas and taking a majority vote, the correct value is obtained, as long as there are at most m malfunctioning sites. The larger the number of replicas, the greater the protection against the information getting lost due to malfunctioning of sites. But this is true only if the probability p of a site malfunctioning is less than 0.5. If $p > 0.5$, replication only reduces the probability of obtaining the correct value. Assume the number of replicas is increased from $2r-1$ to $2r+1$. There are two possible situations in which this addition of two extra replicas makes a difference in determining whether a majority vote yields the correct value or not:

(a) There are $r-1$ malfunctioning sites among the original $2r-1$ sites, but both the additional sites are malfunctioning. Here a majority vote with the original $2r-1$ replicas will yield the correct value, but a majority vote after the addition of the two replicas, will not. The probability of this occurring is

$$p1 = p^2\binom{2r-1}{r-1}p^{r-1}(1-p)^r$$
$$= \binom{2r-1}{r-1}p^{r+1}(1-p)^r.$$

(b) There are r malfunctioning sites among the $2r-1$ original replicating sites, but both the additional replicas are failure-free. Here, while the original set of replicas might not yield the correct value in a majority vote, the augmented set will. The probability of this occurring is

$$p2 = (1-p)^2\binom{2r-1}{r}p^r(1-p)^{r-1}$$
$$= \binom{2r-1}{r-1}p^r(1-p)^{r+1}.$$

Therefore, the improvement in the probability of obtaining the correct value is

$$p2-p1 = \binom{2r-1}{r-1}p^r(1-p)^r(1-2p)$$

which is greater than 0 iff $p<0.5$. Hence replication is desirable only if $p<0.5$. In the rest of this chapter we assume that this condition holds.

3.3. Effect of Malfunctions on Correctness

Let $t: y \leftarrow f(x)$ be a transaction entered into the network which seeks to update y to a new value which is a function of the current value of x. We call x the *read-variable* and y the *write-variable*. Assume we have one copy of x at site X and three copies of y at $Y1$, $Y2$ and $Y3$ (Fig. 3.1). Assume that site X is malfunctioning. Then the values of x or $f(x)$ (depending on where $f(x)$ is computed) sent to $Y1$, $Y2$ and $Y3$ may be incorrect and may be different. If no precautions are taken, the copies of y will take on incorrect and divergent

TRANSACTION Y -- F(X)

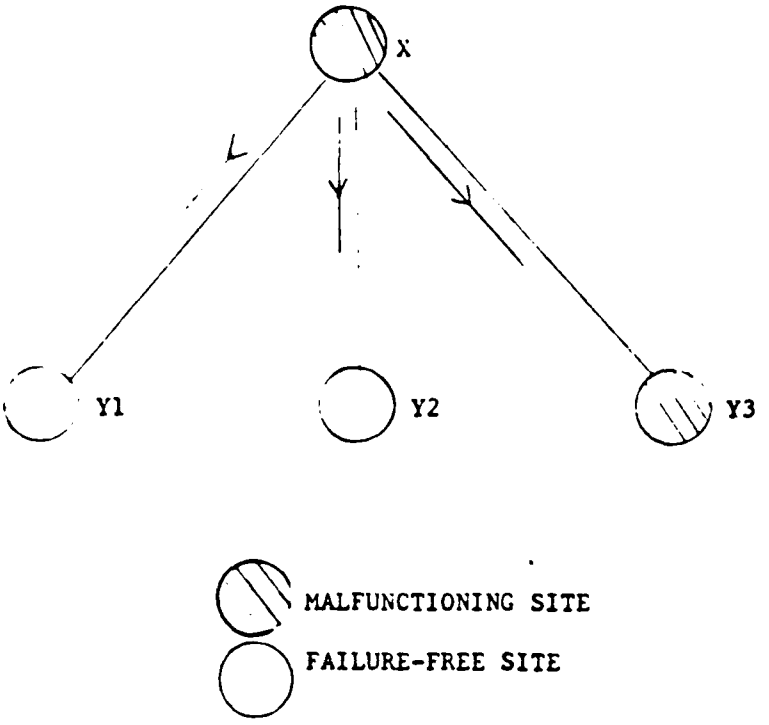


FIG. 3.1. X TRANSMITS THE VALUE OF x TO Y1, Y2 AND Y3.

values. For $Y1$, $Y2$ and $Y3$ to reach agreement is non-trivial, since there may be malfunctioning sites among them too. If other portions of the global information are thereafter updated directly or indirectly as a function of y, the incorrectness of the latter gets propagated. This kind of error propagation, if unchecked, will increasingly disrupt the functioning of the network. To check it, in some cases it will be sufficient if $Y1$, $Y2$ and $Y3$ take on *some common value for* y but in others additional restrictions on this common value will have to be enforced.

As an example to illustrate the importance of maintaining the correctness of global information, consider a dynamic packet radio network in which a group of sites wishes to perform some task, composed of a set of subtasks. Assume that the group has first to determine how many sites are present in the group and how they are connected and then, based on this topology information, to assign subtasks to sites. Assume that for reliability these steps are to be done in a distributed manner and that the following method is chosen. Each site communicates with the rest of the group and determines the topology. Then each of the sites applies a common algorithm to compute the assignment of subtasks to sites. Then we require that a) the correctly operating sites arrive at a common view of the topology so that assignment of subtasks, though done in a distributed manner, is consistent and b) this common view at least "closely" represent the true topology, otherwise the assignment may prove ineffective. (Consider what may happen if a large number of non-existent "ghost" sites are imported into the view by malfunctioning sites. Then critical subtasks may be assigned to ghost sites.)

3.4. Outline of Proposed Approach for Maintaining Correctness

As explained in Section 3.2, we can replicate global information and store the copies at different sites in order to guard its correctness against malfunctioning sites.

Suppose a piece of information is replicated at ($2m+1$) sites, m= 0,1,2... As long as no more than m of these sites malfunction, any site can, by consulting all $2m+1$ sites and taking the majority value obtain the correct value.

In stating that the correct value can be obtained by the above procedure, we are assuming that the following conditions hold: (a) the failure-free sites have the same stored value and (b) this value is correct. However, even if a majority of the $2m+1$ sites are failure-free, conditions (a) or (b) or both could be violated if precautions are not taken when updating the information. This is because of the phenomenon of error propagation explained in Section 3.3.

The problem of preventing error propagation can be stated as follows. Assume that the update $y \leftarrow f(x)$ has been submitted to the system. There are $2t+1$ copies of x each at a different site. This set of sites is called the *transmitter* set $\{T\}$. Similarly there are $2r+1$ copies of y stored in the *receiver* set $\{R\}$. (The sites holding copies of a write-variable in a transaction are called receivers, and the sites holding copies of a read-variable of the transaction are called transmitters. Note that the same site may be a transmitter in one transaction and a receiver in another.) We will assume in this chapter that $\{T\} \cap \{R\}$ = nil, i.e. the sets are disjoint.

In order to prevent error propagation as a result of processing the update, two steps must be taken:

(i) the failure-free sites in $\{R\}$ must reach agreement on the value of x. We call this the *unanimity-reaching* step.

(ii) the value of x agreed on must be verified to be correct. The extent to which this can be done depends on the knowledge that the sites in $\{R\}$ have regarding what values of x are reasonable. This knowledge is stored in the form of *assertions*. We call this step of verifying that x satisfies these assertions the *acceptability-checking* step. The limitation here is that in some cases, it may be difficult to develop assertions that can, to a useful extent, restrict wrong values from passing. In cases where such assertions cannot be generated but it is crucial to protect the updated information, the only solution appears to be to increase the degree of replication of the read-variable and thus diminish the probability of obtaining a wrong value.

We use the the example given above of the group of sites that wish to determine their topology and assign subtasks to different sites in the group. Here the receivers $\{R\}$ are the sites in the group. y is the topology information. x is the position of any given site and $\{T\}$ is the set of sites reporting the position.

In the unanimity-reaching step, the sites in $\{R\}$ reach agreement on the position of a site. The acceptability-checking step can be used to try to screen out "ghost" sites. For instance, the assertion we may require to be satisfied may be that all the sites in some appropriately defined "vicinity" of an alleged site confirm the existence of that site. How effective the assertion is depends on the presence of failure-free sites in the vicinity.

In general, the unanimity-reaching and acceptability-checking steps may be intertwined or follow one another in either order depending on the problem at hand. For example, if the size of x's representation is large, and

if the value received is found to fail the acceptability-checking step a special symbol denoting an unacceptable value may be used in the unanimity-reaching step.

The nature of the acceptability-checking step is very much dependent on the problem at hand. Hence we will not discuss it further. We will discuss the unanimity-reaching step in detail, but first we give a brief description of the results available from the literature that are relevant.

3.5. The Byzantine Generals Agreement Problem

A number of papers have appeared on the so-called Byzantine Generals Agreement (BGA) problem [PEA 80, LAM 80, DOL 81, DOL 82a, DOL 82b, DOL 82c].

Consider a site T which wishes to transmit a value V to a set of receiving sites $\{R\}$. Then the *Byzantine Generals Agreement is reached among* the sites in $\{R\}$ if the following conditions are fulfilled:

1. *If* the transmitter is failure-free, all failure-free receivers agree on V as the common value.

2. All failure-free receivers arrive at the same value, whether the transmitter is malfunctioning or failure-free.

To show the nature of this agreement, we show an example of a network of four sites in Fig. 3.2. Assume that the transmitting site (marked T in the figure) is required to transmit the value 5 to the receiving sites (marked R in the figure). We show two possible situations, in the left and right parts of the figure respectively, the first involving a malfunction in one of the receivers and the second in the transmitter itself. We show how an algorithm given in [LAM 80] is applied to this network to reach the BGA, assuming, as is true for

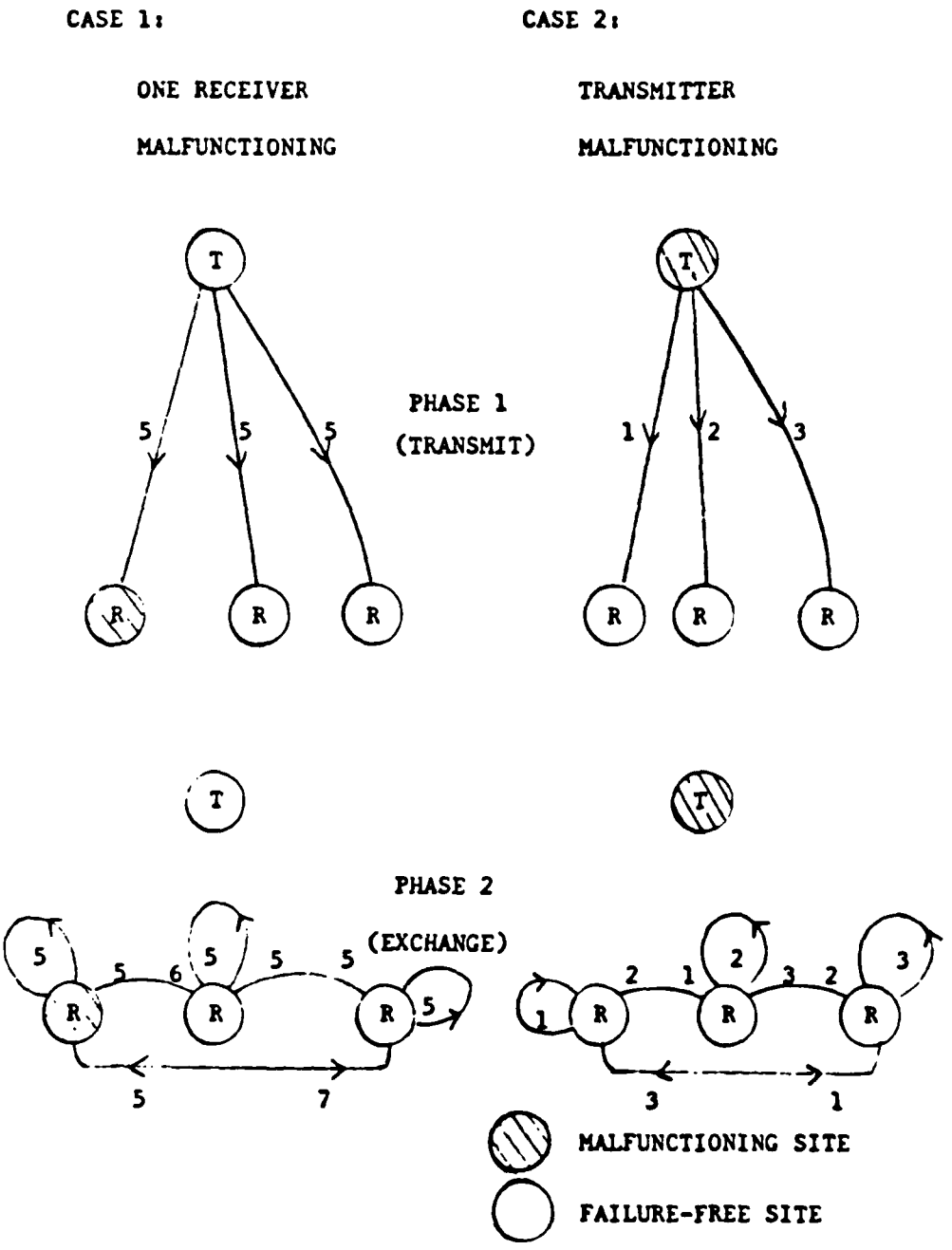


FIG. 3.2. REACHING GBGA IN THE PRESENCE OF ONE MALFUNCTIONING SITE

the two situations described above, that there is at most one malfunctioning site in the network. The algorithm requires two phases of communication in our example, under the assumption made above. In the first phase, the transmitter sends its value i.e. 5 to all the receivers. Note that in the first situation the transmitter is malfunctioning and does this incorrectly. In the second phase, each receiver sends the value received in the first phase to all receivers including itself. Note that in the second situation, one of the receivers is malfunctioning and executes this phase incorrectly. After this phase, each receiver computes the *median* of the values received in the second phase. A quick look at Fig. 3.2 will verify that all correctly operating receivers arrive at the same value, 5 in the first situation, 2 in the second. In the first situation, the transmitter is failure-free and each failure-free receiver has received a majority of values corresponding to the transmitter value. The median computed is thus the transmitter value. In the second situation, the transmitter is malfunctioning. Although the receivers compute a median value which is different from the correct value (5), they all compute the same median value (2). Thus the conditions of BGA are fulfilled in both situations.

The BGA problem has two variants corresponding to whether an authentication facility is available or not. (The example shown above did not use authentication.) Authentication permits a site to *seal* its messages so that another site receiving them can assure itself that their contents have not been altered even though the messages were handled on the way by other sites before reaching it. Thus although a malfunctioning site can abstain from relaying a message which, had it been failure-free it would have relayed, it cannot tamper with its contents and then relay the message without being detected. The authentication facility can be implemented using public-key

encryption [DIF 76,RIV 78].

Consider a network of N sites with at most M malfunctioning sites. It has been shown [PEA 80] that if authentication is not available, it is necessary that $N > 3M$ and whether or not authentication is available, at least $M+1$ phases of communication are required [DOL 82c].

Table 3.1 shows some of the features of the published algorithms to solve the BGA problem. It can be seen that algorithms of polynomial complexity are available for both variants of the BGA problem.

The required number of messages mentioned in the table reflects the worst case. Considering the algorithms which do not employ authentication, the algorithm indicated in the last row has polynomial worst-case communication cost whereas those in the second row (the algorithms indicated in this row are variants of the same basic approach) have exponential worst-case communication costs. But the former algorithm requires more than twice the number of phases and is vastly more complex. Further, the algorithms in the second row can be modified so that while they can still handle upto M malfunctioning sites, they require only about N^2 messages when there are actually no malfunctions [LAM 81a], which will usually be the case. For these reasons they may be preferred to the algorithm indicated in the last row.

We have assumed so far that there is a direct, failure-free link between every pair of sites. In [DOL 81], it is shown how these algorithms can be extended to a point-to-point network, where the connectivity is not complete and the links are not failure-free. Instead of having a bound on the number of malfunctioning sites, a bound is imposed on the number of malfunctioning sites plus failed links. Each message from one site to another is sent along a sufficient number of disjoint routes, so that effectively perfect virtual con-

source	authentication	# of messages	# of phases
PEA 80, LAM 80	yes	$O(N^{M+1})$	$M+1$
PEA 80, LAM 80, DOL 81, DOL 82a	no	$O(N^{M+1})$	$M+1$
DOL 82c	yes	$O(N^2)$	$M+1$
DOL 82c	yes	$O(NM)$	$M+2$
DOL 82c	yes	$O(NM)$	$M+1$
DOL 82b	no	$O(NM+M^3\log M)$ for a 1-bit message	$2M+3$

N =total number of sites

M =bound on total number of malfunctioning sites.

Table 3.1. Algorithms for Byzantine Generals Agreement.

nections are provided between every pair of sites. In presenting our results below, we will continue to assume perfect connectivity, but the techniques of [DOL 81] can be used to relax this assumption.

In [LAM 81a] a scheme for using BGA solutions to implement distributed systems that are able to tolerate malfunctioning sites is described. The basic version of this scheme involves replication of all functions and information at every site in the system. Transactions entering at any point in the system are timestamped and broadcast using BGA techniques so that malfunctioning sites are unable to prevent agreement among failure-free sites as to the transactions received. Control messages exchanged among the sites e.g. for commit processing, also use this reliable broadcast mechanism. Thus all failure-free sites see the same input stream of messages and execute the same sequence of actions. As long as the number of malfunctioning sites satisfies the bounds assumed by the BGA algorithm being used, the system as a whole performs correctly. If the bounds are exceeded, the information stored in the failure-free sites may diverge and from that point on, the system may perform incorrectly until appropriate repair actions are initiated from outside.

In many, if not most, networks, such complete replication would be infeasible since it would require too much storage at each site. It is suggested in [LAM 81a] that in such cases, only critical functions be completely replicated and managed according to the above scheme, constituting a *synchronizing kernel* for the system. Other functions and information would be managed by a separate mechanism. The example of a distributed file system is given as an illustration. Here the directory information and the *open file* and *close file* operations would be in the synchronizing kernel but not the

files themselves or the *read file* and *write file* operations; these would be handled by a separate mechanism.

This separate mechanism would be used to access information at a remote site when it is unavailable locally. Here the danger of error-propagation discussed in Section 3.3 arises, since the remote site may be malfunctioning. Our solution to this problem, as discussed in Section 3.4 consists of a unanimity-reaching step and an acceptability-checking step. For the unanimity-reaching step, the BGA problem is relevant. However, we may have a multiplicity of transmitters (since the information is replicated, though partially) whereas the BGA has been stated for the case of one transmitter. Also it may be too expensive to use BGA techniques on each remote access to global information. We discuss these issues in Section 3.6.

3.6. Details of Proposed Approach

3.6.1. The Generalized Byzantine Generals Agreement Problem

The requirements of the unanimity-reaching step of Section 3.4 may be stated as follows.

Given a set of transmitters $\{T\}$ each of which has a copy of a given piece of information (the read-variable), and a set of receivers $\{R\}$ (which hold copies of the write-variable) which wish to access this information, the Generalized Byzantine Generals Agreement (GBGA) is reached by the receivers if

a) All failure-free receivers agree on the same value.

b) If a majority of transmitters are failure-free, and each of the transmitters in this majority has the same value V for the information, then the receivers agree on V as the common value.

Clause a) is the same as for the BGA. The changes for clause b) are simple to understand. If a majority of transmitters are not failure-free, i.e. if a majority of malfunctioning sites exists, the latter by acting in collusion can make it impossible to deduce that the remaining minority of transmitters are the ones from which the correct value is to be obtained. The reason for requiring that the failure-free site majority of transmitters should have the same value is that there is a possibility that they may have divergent values because of prior error propagation, in which case, clause a) already specifies the best the receivers can hope to do.

3.6.2. Malfunction-Tolerance Specification

If we wish to use GBGA to process an update transaction $y \leftarrow f(x)$ then we must specify the bounds on the number of malfunctioning sites called the *malfunction-tolerance specification, (MTS)*. The protocols used in processing the update will then be such that as long as the actual number of malfunctioning sites is within the MTS, GBGA will be reached by the receivers.

Assume that there are $(2t+1)$ sites, forming the set of transmitters $\{T\}$, which hold copies of x and $(2r+1)$ sites forming the set of receivers $\{R\}$ holding copies of y. $F(S)$ denotes the number of malfunctioning sites in the set S. Examples of possible MTSs are:

(a) $F(\{T\})=0$

(b) $F(\{T\})\leq 1$

(c) $F(\{T\})\leq t$

(d) $F(\{R\})\leq r$

(e) $F(\{R\}\cup\{T\})\leq r$, etc.

These MTSs specify different degrees of malfunction-tolerance and the protocols that achieve GBGA given these MTSs have different communication

and computation costs associated with them. For example, for MTS (a), any single transmitter can be accessed to get the value of x; for MTS (b), three transmitters can be accessed (assuming $|T| \geq 3$) and the majority value taken, and so on. Thus a tradeoff exists between these costs and the degree of malfunction-tolerance obtained.

3.6.3. Scheme Specification

The global information consists of a set of data items. Suppose we are given the update interactions between them in the form of a set of ordered pairs (x_i, x_j) where the existence of a pair (x_i, x_j) implies the existence of an update interaction $x_i \leftarrow f_{x_i x_j}(x_j)$. [We assume update interactions of this form for simplicity.] Suppose we are also given the degree of replication for each item x_j.

The *scheme specification (SS)* specifies a protocol for each update interaction pair. For example, SS could specify that for a given pair (x_i, x_j) the protocol for this interaction should achieve GBGA with the MTS $F(\{R\}) \leq r$. (Here the degree of replication of x_i is $2r+1$.)

Given the data items, the update interactions, the degrees of replication and the scheme specification, the behavior of the system information as to its correctness characteristics in the presence of malfunctioning sites (including the error propagation effects) can be deduced. Hence in order to obtain the desired behavior, the degrees of replication and the scheme specification should be chosen appropriately. Below we give two examples of possible scheme specifications.

3.6.3.1. Scheme Specification A

In Section 3.4, we commented that the correct value of a piece of information y could be obtained as long as a majority of the sites holding copies of it were failure-free, provided they were not contaminated by error propagation. This contamination occurs when the information is being used as a write-variable, i.e. when the sites holding copies of it are acting as receivers in a transaction $y \leftarrow f(x)$. If the failure-free sites among the sites holding copies of y are a minority, the correct value of y would not be available and there would be no point in trying to ensure that these failure-free sites arrive at the same value of the read-variable x (or $f(x)$ if the transmitters do the computation of $f(x)$). Hence it is reasonable to use the following scheme specification:

SS A: GBGA must be reached in $\{R\}$ whenever $F(\{R\}) \leq r$.

SS A achieves the extreme of absolutely no error propagation in the following sense. Consider a chain of update interaction pairs: ($\ldots x_i \leftarrow x_i \leftarrow x_j \leftarrow x_k \leftarrow \ldots$.) [i.e. x_i is updated with x_i as read-variable, x_i is updated with x_j as read-variable in turn, etc.]. Suppose the malfunctioning sites among those sites which hold copies of x_j constitute a majority. As a result of the particular scheme specification used here, x_i will not get contaminated by the update interaction $x_i \leftarrow x_j$. Hence, as long as the sites holding copies of any item x_i have a failure-free majority, x_i will be correct, since it cannot be contaminated by error-propagation.

SS A can be implemented by the following algorithm:

Alg:

(1) Each member of $\{R\}$ samples each member of $\{T\}$ and computes the

majority of the $|T|$ values received.

(2) Each member of $\{R\}$ broadcasts the value obtained in step (1) to all the sites in $\{R\}$ using a BGA algorithm with a bound r on the number of malfunctioning sites in $\{R\}$.

(3) Each member of $\{R\}$ computes the median of the $|R|$ values received in step (2).

Note that the median of a set of values coincides with the majority value, if one exists, and is unique for a given set of values. Thus if each receiver receives the same set of values (which may not have a majority value), or if all receivers receive sets of values having a common majority value, then computing the median as the final value ensures unanimity.

As mentioned in Section 3.5 BGA has two variants. When authentication is not available, step (2) can be executed only if $|R|$ is at least $3r+1$. Only $2r+1$ copies of a piece of information are required to preserve its availability in the presence of upto r malfunctioning sites. Hence there must be r extra sites among the receivers, which need not have physical copies of the information, but which take part in the algorithm described above (in Fig. 3.3(a), they are shown as having "phantom" copies). At the end, the sites with physical copies update them to the median value computed in step 3. "$\{R\}$" in the scheme specification must be taken to mean these $3r+1$ sites out of which only $2r+1$ actually have copies of the information and take part in transmitting it when it is used as read-variable in some update transaction. As shown in Fig. 3.3(b), no phantom copies are required when authentication is available.

For both variants, $r+2$ phases of communication are required. Hence this scheme specification would be prohibitively expensive in the amount of

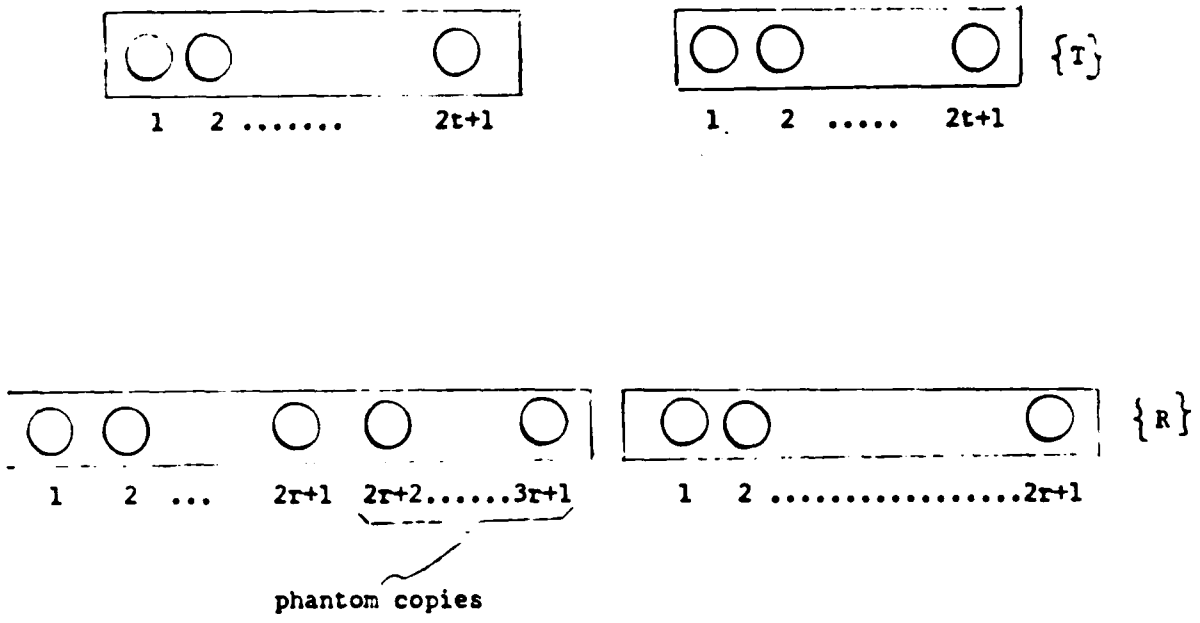


FIG. 3.3. EFFECT OF USE OF AUTHENTICATION FACILITY ON RECEIVER SET CONFIGURATION IN SCHEME SPECIFICATION A.

communication required and the *time* taken to process an update. Thus, it is feasible to use it only if updates are very rare and the need to protect the information is critical.

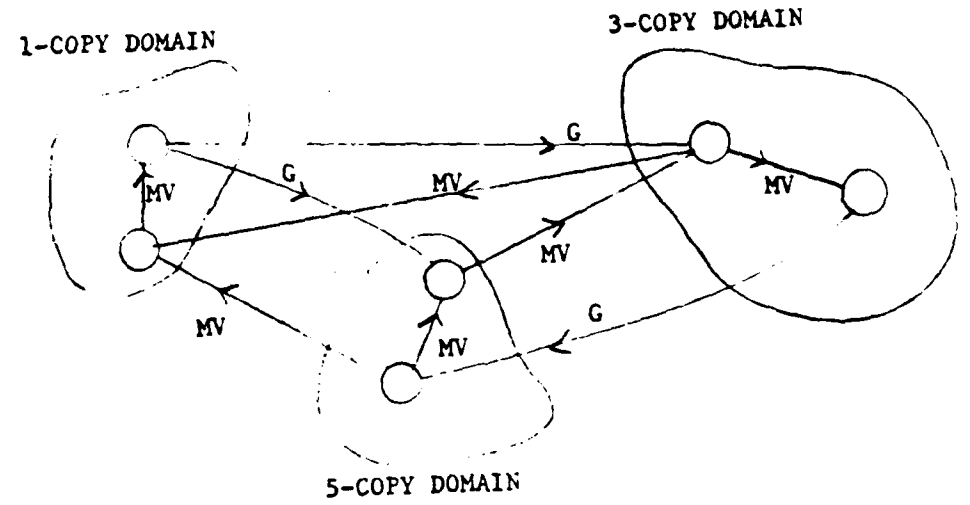
Below we present another scheme to remedy the drawbacks of *SS A*.

3.6.3.2. Scheme Specification B

SS B: as in *SS A*, except that when $|T| \geq |R|$, majority voting on the values sent out by the transmitters is used by every member of $\{R\}$ to get the value of the read-variable.

Conceptually, the global information can be divided into domains depending on the degree of replication, e.g. a 1-copy domain, a 3-copy domain, a 5-copy domain,etc (Fig. 3.4). *SS B* has the property that contamination can spread within a given domain and into lower-order domains but not into higher-order domains. By properly allocating the global information to the domains, the number of updates which require GBGA protocols can be reduced and thus the number of updates incurring the high communication costs and processing time typical of *SS A*. A critical piece of information should be placed in a higher-order domain so that it is less likely to become unavailable as a result of a majority of the sites that hold copies of it malfunctioning. By the same argument, a less critical piece of information should be placed in a lower-order domain. Thus it is more likely to become unavailable. However, it is prevented from contaminating the more critical information by the protocol which governs such interactions.

For example, in a banking application, all information relating to accounts larger than or equal to α dollars could be placed in the 3-copy domain, and information relating to accounts smaller than α dollars could be placed in the 1-copy domain. Most transactions would be limited to a single



MV → SIMPLE MAJORITY VOTE

G → GBGA WITH MTS $F(R) \leq r$

FIG. 3.4. SCHEME SPECIFICATION B

domain. A funds transfer from an account in the 1-copy domain to another in the 3-copy domain would cause the GBGA protocol to be invoked in updating the latter.

3.7. Intermediate Cost Protocols

3.7.1. Motivation

Consider the following MTS:

$M1$: $F(\{T\}) \leq t$ with all failure-free transmitters having the same value.

Reaching GBGA under $M1$ can be done simply and inexpensively by having each receiver take the median of the values of all the transmitters. But consider the update chain $\ldots x_h \leftarrow x_i \leftarrow x_j \leftarrow x_k \leftarrow x_l \ldots$. If for any of the variables, the number of malfunctioning replicas is a majority, error can propagate backwards along the chain. This MTS is used in SIFT [WEN 78].

Consider next the following MTS:

$M2$: $F(\{R\}) \leq r$.

This is the MTS used in *SS A*. Although reaching GBGA under $M2$ is expensive as mentioned in Section 3.6.3.1, there is no error propagation.

Now consider the MTS:

$M3$: $F(\{T\} \cup \{R\}) \leq r$, with all failure-free transmitters having the same value.

It will be shown below that to reach GBGA for this MTS, it requires an algorithm whose costs are a function of the difference in the degree of replication of the read and write-variables decreasing as the difference decreases. This is an appealing property, for as we noted in Section 3.2, the degree of replication is a measure of the probability of the information becoming unavailable because a majority of the sites holding copies of it are malfunctioning. One would like to be more careful (at the expense of higher incurred

costs) in interacting with information that is more likely to be incorrect. This MTS facilitates this. However, it permits error-propagation because of the following reason. Considering any update in the chain mentioned above, if the bounds specified on the number of malfunctioning transmitters and receivers is exceeded, GBGA may not be reached among the copies of the variable updated. Then, considering the preceding update in the chain, the condition in the MTS specifying that all failure-free transmitters should have the same value is not satisfied. Hence GBGA may not be reached for this update, even though the bound on the number of malfunctioning transmitters and receivers for this update is satisfied. In this way error can propagate backwards along the chain. Therefore, at appropriate points on the chain, an MTS which prevents error propagation e.g. $M2$, should be used, and between these points MTSs such as $M1$ and $M3$ could be used.

In Section 3.7.2, the minimum total number of sites required to reach GBGA without authentication under MTS $M3$ is determined. In Sections 3.7.3 and Sections 3.7.4, algorithms for reaching GBGA without and with authentication under this MTS are developed.

3.7.2. Minimum Number of Sites for GBGA under MTS M3

Consider the following situation. A network has N sites of which a set $\{T\}$, $|T| = 2t+1 < N$ has a value to transmit to the rest of the sites in the network which are the receivers. It is assumed that all failure-free transmitters have the same value. The number of malfunctioning sites in the network $\leq m$. The problem is to find the minimum value of N, N_{min}, such that GBGA can be reached among the receivers. If $t \geq m$ or there is only one receiver, then a simple majority vote solves the problem. Hence, from now on we only concern ourselves with the case where $t < m$ and there is more than one

receiver.

It is clear that $N_{min} \leq 3m+1$, for if $N \geq 3m+1$, each transmitter can send its value to all the sites in the network using a BGA algorithm parameterized for a bound of m malfunctioning sites. Then each receiver can take the median of the $2t+1$ values received to be the value of the transmitters and GBGA will be reached. [Note however that the costs of reaching GBGA with this procedure are *not* a function of the difference in the number of transmitters and the number of receivers. Algorithms which do have this property will be presented in Sections 3.7.3. and 3.7.4.]

We show that the above bound is tight, i.e. $N_{min} = 3m+1$. For this purpose, we use the concept of *scenarios* introduced in [PEA 80]. Our proof is a non-trivial extension of the proof given in [PEA 80] to establish that at least $3m+1$ sites are required to reach BGA in a network with at most m malfunctioning sites, if authentication is not used.

Let $\{P\}$ be the set of sites in the network, and define a scenario S as a mapping from the set of non-empty strings W over $\{P\}$ and ending with a transmitter, to V, the set of values. For a given r in $\{R\}$, define the r-scenario S_r corresponding to a scenario S as a restriction of the mapping S to strings in W beginning with r.

Let $\{X\}$ be a set of sites in the network which are failure-free. A scenario S is consistent with $\{X\}$, if for each $q \in \{X\}$, $p \in \{P\}$, $w \in W$, $S(pqw)=S(qw)$, i.e. each site in $\{X\}$ always relays values it receives correctly.

For each r in $\{R\} = \{P\}-\{T\}$, let F_r be a mapping that takes a r-scenario S_r and returns a value in V, which is the value of the transmitters arrived at by r finally. In order for $\{F_r | r \in \{R\}\}$ to provide GBGA for each scenario S consistent with some set of sites $\{X\}, |X| \geq N-m$, we must have

(A1) if a majority of transmitters are failure-free and have a value v_i, $F_\tau(S_\tau)=v_i$ for all $\tau \varepsilon \{R\} \cap \{X\}$.

(A2) for $p,q \varepsilon \{R\} \cap \{X\}$, $F_p(S_p)=F_q(S_q)$.

Suppose $N \leq 3m$ and if possible let $\{F_\tau \mid \tau \varepsilon \{R\}\}$ provide GBGA. Divide $\{R\}$ into three sets A,B,C each having at most m sites, with A having a majority of transmitters, B having the remaining transmitters, and both B and C having at least one receiver(Fig. 3.5). This division is possible since $t<m$ and there are at least two receivers. Let v and v' be two distinct values in V. Define the scenarios $S1,S2,S3$ consistent with $A \cup B, A \cup C, B \cup C$ as shown below. In the following specifications, a_i and b_i represent any transmitter in A and B respectively, a,b,c represent any sites in A,B,C respectively and w represents any string in W.

$S1$:

$$S1(aa_i) = S1(ba_i) = S1(ca_i)$$
$$= S1(ab_i) = S1(bb_i) = S1(cb_i) = v.$$

$$S1(aaw) = S1(aw) \quad S1(abw) = S1(bw)$$
$$S1(baw) = S1(aw) \quad S1(bbw) = S1(bw)$$
$$S1(caw) = S1(aw) \quad S1(cbw) = S1(bw)$$

$$S1(ccw) = S1(cw)$$
$$S1(acw) = S1(cw)$$
$$S1(bcw) = S3(cw)$$

$S2$:

$$S2(aa_i) = S2(ba_i) = S2(ca_i) = v'$$
$$S2(ab_i) = S2(bb_i) = S2(cb_i) = v.$$

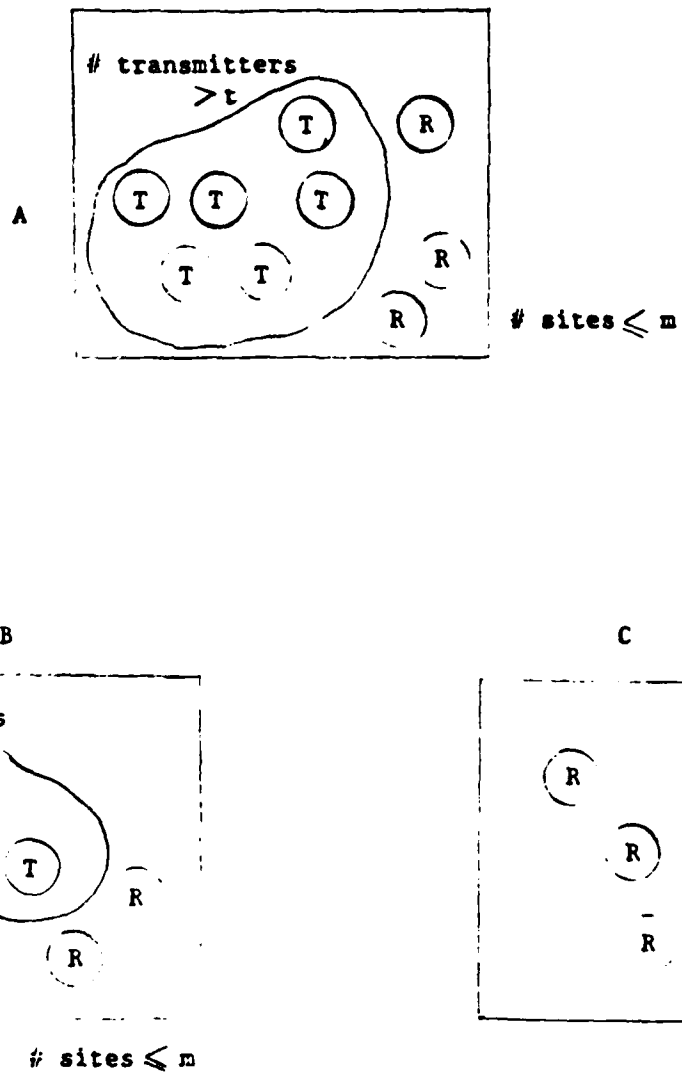


FIG. 3.5. NETWORK CONFIGURATION FOR SHOWING THAT $N_{min} > 3m$.

$$S2(aaw) = S2(aw) \quad S2(abw) = S2(bw)$$
$$S2(baw) = S2(aw) \quad S2(bbw) = S2(bw)$$
$$S2(caw) = S2(aw) \quad S2(cbw) = S3(bw)$$

$$S2(acw) = S2(cw)$$
$$S2(bcw) = S2(cw)$$
$$S2(ccw) = S2(cw)$$

$S3$:

$$S3(aa_i) = S3(ba_i) = v \quad S3(ca_i) = v'$$
$$S3(ab_i) = S3(bb_i) = S3(cb_i) = v.$$

$$S3(aaw) = S3(aw) \quad S3(abw) = S3(bw)$$
$$S3(baw) = S1(aw) \quad S3(bbw) = S3(bw)$$
$$S3(caw) = S2(aw) \quad S3(cbw) = S3(bw)$$

$$S3(acw) = S3(cw)$$
$$S3(bcw) = S3(cw)$$
$$S3(ccw) = S3(cw)$$

Next we show that

E:(i) $S3(bw) = S1(bw)$

(ii) $S3(cw) = S2(cw)$

E is true when w is of length 1. This follows directly from the fact that w is then either a_i or b_i and from the scenario specifications.

Assume E true for $|w| \leq l$. Let w_l be a string in W of length l.

(a)From the scenario specifications, we have

$$S3(baw_l) = S1(aw_l) \text{ and}$$
$$S1(baw_l) = S1(aw_l).$$

Therefore

$$S3(ba w_i) = S1(ba w_i).$$

(b)From the scenario specifications, we have

$$S3(bb w_i) = S3(b w_i) \text{ and}$$

$$S1(bb w_i) = S1(b w_i).$$

By our inductive assumption,

$$S3(b w_i) = S1(b w_i).$$

Therefore

$$S3(bb w_i) = S1(bb w_i).$$

(c)From the scenario specifications, we have

$$S3(bc w_i) = S3(c w_i) \text{ and}$$

$$S1(bc w_i) = S3(c w_i).$$

Therefore

$$S3(bc w_i) = S1(bc w_i).$$

From the above it follows that $S3(bw) = S1(bw)$ for $|w| \leq l+1$. Similarly $S3(cw) = S2(cw)$ for $|w| \leq l+1$.

Thus E is true for $|w| \leq l+1$, and therefore for all $|w|$.

Let b_r and c_r be receivers in B and C respectively. Then $S3_{b_r} = S1_{b_r}$ and $S3_{c_r} = S2_{c_r}$.

By A1,

$$F_{b_r}(S3_{b_r}) = F_{b_r}(S1_{b_r}) = v \text{ and}$$

$$F_{c_r}(S3_{c_r}) = F_{c_r}(S2_{c_r}) = v'.$$

By A2,

$$F_{b_r}(S3_{b_r}) = F_{c_r}(S3_{c_r})$$

implying $v = v'$. This contradicts our earlier assumption. Hence $N_{min} = 3m+1$.

3.7.3. Implementing GBGA under MTS M3 without Authentication

The previous section shows that if GBGA is to be reached without using authentication under the MTS

$M3: F(\{T\} \cup \{R\}) \leq r$ with all transmitters having the same value,

then, if $t < r$ (where $|T| = 2t+1$), a minimum of $3r+1$ sites is required. The updated variable is replicated at $2r+1$ sites. Hence a number of "phantom" receivers equal to $max(0,(2r+1)+(2t+1)-(3r+1)) = max(0,2t+1-r)$ will be required. $\{R\}$ is then the set of receivers $\{R_r\}$ which replicate the updated variable plus the set of phantom receivers $\{R_p\}$. However, the procedure given in the previous section for reaching GBGA using this configuration requires $r+1$ phases (and hence, as mentioned there, its costs are not a function of $r-t$).

As mentioned in Section 3.7.1, it is possible to construct GBGA algorithms for MTS $M3$ using fewer phases. This is done by using BGA algorithms in a different manner from that in the previous section. For the reasons mentioned in Section 3.5, we choose the BGA algorithm BG1 described in [PEA 80,LAM 80,DOL 81,DOL 82a] as the kernel of the GBGA algorithm described below.

Consider a network consisting of a transmitter T and a set of receivers $\{R\}$ with $|R| \geq 3m$. It is required to have the receivers reach BGA on the value of T as long as the total number of malfunctioning sites in the network $\leq m$. The algorithm BG1 is as follows:

Algorithm BG1($\{R\}$, m):

(1) The transmitter T sends its value to every receiver in $\{R\}$.

(2) If $m > 0$ then

(a) for every $r \in \{R\}$, let v_r be the value receiver r has obtained in step 1.

Receiver r acts as the transmitter in the algorithm BG1($\{R\}-r, m-1$) to send the value v_r to every other receiver in $\{R\}-r$.

(b) for every $r' \in \{R\}$ and each $r \neq r'$ in $\{R\}$, let $v_r(r)$ be the value receiver r' receives from receiver r in step 2a. If no value is received, set $v_r(r)$ to 0. Let $v_r(r')$ be the value receiver r' has received from transmitter T in step 1. Receiver r' determines the value of the transmitter as $median\{v_r(x) | x \in \{R\}\}$.

Now, we show how to use algorithm BG1 to implement GBGA under MTS $M3$ with fewer phases than $r+1$.

Consider a network with $|T| = 2t+1$ transmitters, with all failure-free transmitters having the same value, and with at most r ($r > t$) malfunctioning sites in the network. Let the remaining sites in the network $\{R\}$ be of number $|R| \geq 3r-t$.

Algorithm GBG1:

(1) Each transmitter in $\{T\}$ sends its value to a designated subset of receivers $\{R_r\}$, $|R_r| = 2r+1 \leq |R|$.

(2) Each receiver r_r in $\{R_r\}$ computes the median of the $(2t+1)$ values received in step 1 to obtain v_{r_r}.

(3) Each receiver r_r in $\{R_r\}$ broadcasts its value v_{r_r} to every other receiver using BG1($\{R\}-r_r, r-t-1$).

(4) Let $\{X_r\}$ be the set of $2r$ values received in step 3 and the single value computed in step 2 by the receiver r in $\{R\}$. It computes the value of the transmitters as $median\ \{X_r\}$.

Thm 3.1: Algorithm GBG1 provides GBGA under MTS $M3$ in $r-t+1$ phases.

Proof: Steps 1 and 2 together involve one phase of communication. Steps 3

and 4 involve $(r-t-1)+1$ or $(r-t)$ phases. Hence GBG1 involves $r-t+1$ phases in all.

Case 1: A majority of failure-free transmitters does not exist. Then GBGA requires each failure-free receiver to arrive at the same final value. Since there are at least $t+1$ malfunctioning transmitters, there are at most $r-t-1$ malfunctioning receivers. From the correctness of BG1, it follows that every failure-free receiver r has the same set of values $\{X_r\}$ in step 4 and hence unanimity is reached.

Case 2: A majority of failure-free transmitters exists and v is their common value. Then GBGA requires each failure-free receiver to arrive at the final value of v. Let $\{CR_r\}$ be the set of failure-free receivers in $\{R_r\}$. $|CR_r| \geq r+1$ since $|R_r|=2r+1$ and there are at most r malfunctioning sites in the network.

Each site cr_r in $\{CR_r\}$ computes the value v in step 2. We claim that v will be the value received from cr_r by every failure-free receiver in step 3. Then the final value computed in step 4 will be v.

The proof of our claim is based on a lemma given in [DOL 82a] for the BGA algorithm BG1. This lemma states that, in a network with a single transmitter T', a set of receivers $\{R'\}$ with at most m malfunctioning sites, BG1($\{R'\}, x$) provides BGA if the transmitter is failure-free and $|R'| \geq 2m+x$. In step 3 of GBG1, as executed by cr_r, we have cr_r as a failure-free transmitter, executing BG with $x=r-t-1$. The set of receivers it is transmitting to, has cardinality $3r-t-1=2r+(r-t-1)$. Hence our claim is proved.

For completeness, we give below the proof of the above-mentioned lemma. Consider a network consisting of a failure-free transmitter T' with a value v', and a set of receivers $\{R'\}$, $|R'| \geq 2m+x$ with at most m malfunction-

ing sites. To prove that BG1($\{R'\}$, x) produces BGA, we use induction on the value of x. If $x=0$, the final value arrived at by each failure-free receiver is the value received in step 1 of BG1 from the transmitter, namely v'. Hence BGA is reached for $x=0$.

Assume the lemma holds for $x=k(\geq 0)$. Consider $x=k+1$. In step 1, each failure-free receiver, r' receives the value v'. In step 2a, it applies the algorithm BG1($\{R'\}-r'$, k) to send the value v' to all the receivers in the set $\{R'\}-r'$ which contains at least $2m+k$ sites, and hence the induction hypothesis implies that every other failure-free receiver obtains from r' the value v'. The set $\{R'\}$ contains at least $2m+k+1$ sites. Since $k\geq 0$, and there are at most m malfunctioning sites in $\{R'\}$, every failure-free receiver computes a final value v' in step 2b. This proves the lemma.

—

Using the algorithm GBG1, we can implement GBGA with MTS $M3$ in $r-t+1$ phases and with $(3r-t)-(2r+1) = r-t-1$ phantom receivers. Thus, this algorithm reduces the communication overhead and number of phases to a function of the *difference* in the degrees of (physical) replication of the read- and write-variables. As mentioned in Section 3.7.1., this is a desirable property.

3.7.4. Implementing GBGA under MTS M3 using authentication

A similar reduction in the number of phases can be realized in constructing an algorithm for reaching GBGA under MTS $M3$ using authentication. The use of authentication sharply reduces the number of messages needed. Our algorithm uses as its kernel a BGA algorithm BG2 suggested in [DOL 82c] that uses authentication.

Consider a network consisting of a transmitter T and a set of receivers $\{R\}$ which has at most m malfunctioning sites.

Algorithm BG2:

(1) The transmitter T signs and sends its value to all receivers.

(2) Each receiver waits for the receipt of messages. If during phase k, a receiver receives a message containing value v and signed by k distinct sites (beginning with the transmitter), then the receiver inserts v in its list of received values if not already in it and if the list does not already contain two values. If the value v gets inserted, and $k<m+1$ then the receiver signs the message and sends it to all receivers, whose signatures are not in the message, in phase $k+1$.

(3) After phase $m+1$, if a receiver has exactly one member in its list of received values, then that is chosen as the final value, otherwise it agrees on a default value.

This algorithm requires $O(N^2)$ messages, where N is the total number of receivers.

We use BG2 to construct an algorithm GBG2 providing GBGA under MTS $M3$. Consider a network consisting of a set of transmitters $\{T\}$, $|T|=2t+1$ and a set of receivers $\{R\}$, $|R|=2r+1$, $t<r$, $F(\{T\}\cup\{R\})\leq r$, and all the failure-free transmitters have the same value.

Algorithm GBG2:

(1) Each transmitter sends its value to all receivers.

(2) Each receiver computes the median of the $2t+1$ values received.

(3) Each receiver acts as transmitter of the value computed in step 2 to all other receivers using algorithm BG2 parameterized for a maximum of $r-t-1$

malfunctioning receivers.

(4) Each receiver computes the *median* of the $2r$ values arrived at in step 3 and the value computed in step 2. This median is the final value agreed on.

Thm. 3.2: Algorithm GBG2 provides GBGA under MTS $M3$ in $r-t+1$ phases.

Proof: Steps 1 and 2 contribute one phase and steps 3 and 4 contribute $r-t$ phases. Hence a total of $r-t+1$ phases is required.

Case 1: A majority of malfunctioning transmitters exists. Therefore there are at most $r-t-1$ malfunctioning receivers. Hence, by the correctness of BG2, each failure-free receiver has the same set of $2r+1$ values whose median it computes in step 4. Therefore, all failure-free receivers unanimously agree on some value.

Case 2: A majority of failure-free transmitters exists and they have the common value v. In step 2, each failure-free receiver r computes the value v, and in the first phase of step 3, sends this value, signed, to all other receivers. Moreover, since this is the only value it signs as transmitter in step 3, each failure-free receiver will agree on v as the value transmitted by r in step 3. Since a majority of receivers is failure-free, each receiver will compute v as the final value in step 4.

—

For this algorithm again, the communication overhead and the number of phases is a function of the difference in the degrees of replication of the read- and write- variables.

3.8. Conclusion

In this chapter, we introduced the correctness aspect of the availability attribute of global information. Correctness becomes important when tolerance to malfunctioning sites is required. We advanced some theoretical considerations for dealing with this mode of failure. It is clear from the above discussion that the GBGA can be used in a flexible manner to obtain the required degree of tolerance of malfunctions. GBGA protocols can also be mixed with cheaper protocols which provide a lower degree of tolerance in such a way that meaningful guarantees can be given as to the correctness of the global information.

It is evident that malfunctions are expensive to cope with. But the malfunction as a model of failure is appealing since it represents the worst kind of faulty behavior a site can exhibit. It would require very complex reasoning to show the probability of hardware or software failures which could result in a site malfunction to be small enough to ignore. Further even if such reasoning could be provided, a site could be taken over by a malicious agent. For these reasons, we believe that techniques must be developed to deal with malfunctions. One possible approach around the difficulty of the high costs of such techniques is to attempt to use them sparingly and selectively and we have explored this approach in this chapter.

In our approach, the information is selectively replicated to different degrees, depending on its criticality. If, in spite of this replication, the correct value of some of this information becomes unavailable due to the malfunctioning of a number of sites, we try to prevent updates from propagating error to information whose correct value is still available. A combination of an acceptability-checking step in which assertions are used to weed

out wrong values of the read-variable, and a unanimity-reaching step (whose requirements are formalized in the GBGA) which provides consensus on the value of the read-variable, is used for this purpose. We presented a variety of protocols which achieve GBGA for different kinds of bounds on the number of malfunctioning sites. These protocols differ in the amount of malfunction-tolerance they provide and in their associated costs, and thus allow the designer to make appropriate tradeoffs.

CHAPTER 4

DEADLOCK DETECTION IN DISTRIBUTED DATABASE SYSTEMS

4.1. Introduction

In this chapter, we present centralized and distributed algorithms for deadlock detection in distributed database systems. These algorithms use a clock facility to ensure that deadlocks indicated really exist and that no existing deadlocks go undetected.

In Section 4.2, the various approaches available for deadlock handling are discussed. In Section 4.3, race conditions that complicate deadlock detection in distributed systems are discussed. Section 4.4 introduces terminology and lists some assumptions. In Sections 4.5 and 4.6, centralized and distributed schemes for deadlock detection are presented, along with past work in the area.

4.2. Approaches to Deadlock Handling

Deadlock may be described as a situation of mutual wait among a set of blocked processes, each of which is waiting to acquire one or more resources held by other processes in the set. The easiest approach to deal with deadlock is to use timeouts to abort any process that has been waiting too long (or to abort the process that has been causing another to wait too long). Though this approach is feasible for lightly loaded systems in which contention is rare, it runs into difficulties in congested situations. At such times, timers will run out often causing many processes to be aborted, and prolonging the congestion [GRA 78]. Other deficiencies of this approach are

cyclic restart or *livelock* [ISL 80], and wastage of resources arising from aborted computations.

Three approaches have been developed to deal with the deadlock problem: *prevention*, *avoidance* and *detection*.

In deadlock prevention techniques, the requests for resources are constrained to occur in particular ways so that deadlocks never occur. Such techniques include requesting all resources needed by the process at once, imposing a total ordering on the resources and requesting needed resources in this order, the WOUND-WAIT and WAIT-DIE algorithms of [ROS 77]. These techniques restrict the amount of concurrency as a result of the constraints they impose. Further, the first two approaches are not appropriate for database systems, since it is not always possible to predict ahead of time which resources will be needed by a process.

Deadlock avoidance techniques permit the granting of a requested resource only if to do so would still allow all processes at least one way to complete execution. Habermann's algorithm [HAB 66] is the best-known deadlock avoidance scheme. Since a worst case scenario for future requests is assumed in determining if a resource grant is safe, concurrency is still restricted. The deficiency of having to know ahead of time which resources are going to be required is also present in avoidance schemes. Further, in a distributed system, the computation of whether a resource grant is safe or not, requires knowledge of the states of the various processes at the various sites in the system. Hence it is difficult to do resource allocation in an efficient yet decentralized manner.

Deadlock detection techniques allow a maximum of concurrency by granting resources whenever they are available. At appropriate times, the

status of resources and processes in the system is examined to see if a deadlock exists. In order to do so, this status is maintained in the form of a graph in which the nodes represent processes and resources, the process-to-resource arcs represent outstanding requests and the resource-to-process arcs represent possession of resources by processes. The necessary and sufficient conditions for deadlock in systems containing *reusable resources*, e.g., files, memory, etc. and/or *consumable resources*, e.g., messages, have been developed in [HOL 72]. In the case of distributed databases, under the assumption that all resources are one-of-a-kind and that a process must wait till *all* resources it has requested have been granted before it can proceed, the necessary and sufficient condition is the existence of a cycle in the process-resource graph.

Deadlock detection in a distributed system may be done in a centralized, hierarchical or distributed manner. In centralized detection, a single site is designated as the deadlock detector. It collects the status of processes and resources in the system and checks for deadlocks in the assembled graph. (Detection of deadlocks confined to one site may be done locally.) The disadvantage of this method is its vulnerability to failure of the deadlock-detecting site. Also, if the network is large, the load imposed on the deadlock-detecting site may be too large.

Both of the above problems are ameliorated in the hierarchical method. Here, the sites are partitioned into a hierarchy of clusters, with each cluster having a deadlock detector site. A deadlock confined to sites within a cluster is detected by the local deadlock detector; a deadlock spanning multiple clusters is detected by the deadlock detector in the lowest cluster which is a parent of all the clusters involved. Here, as in the centralized case, detec-

tion of a deadlock may be delayed by the failure of sites other than the deadlocked sites. There is also the problem of choosing the clusters appropriately in order that most of the deadlock computation may be done at the local cluster level instead of having to refer to higher levels in the hierarchy.

In the distributed scheme, the deadlock detecting facility is distributed equally among all the sites in the network. In general, distributed schemes involve more communication overhead than the centralized schemes. This happens because in distributed schemes graph traversals initiated at different points in the process-resource graph in order to check for deadlocks, go over the same portions of the graph. This repetition is to some extent unavoidable in a distributed algorithm. The advantages of distribution are that the detection of a deadlock involves only the sites involved in the deadlock. Hence, the vulnerability to failures of the designated deadlock detecting sites which characterizes the centralized and hierarchical schemes is not present here.

4.3. Race Conditions in Deadlock Detection

In a single computer system, the deadlock detector can stop all activity in the computer, while it examines the necessary tables, queues, etc., to construct the process-resource graph. In the case of a distributed system, it is not feasible to stop the entire system in order to take a similar snapshot of the processes and resources in the system and the messages that may be in transit. Therefore the status of the processes and resources at each site must be recorded asynchronously and the global status computed in a consistent manner from these recordings. A complicating factor here is that messages may take arbitrary periods of time to reach their destinations.

As an illustration of the problems involved, consider two sites $S1$ and $S2$ of a network in which a third site DD acts as a centralized deadlock detector. Suppose process $P1$ and resource $R1$ reside at site $S1$ and process $P2$ and resource $R2$ reside at site $S2$ (Fig. 4.1). Initially $P1$ and $P2$ request and acquire resources $R1$ and $R2$ respectively. Next $P2$ sends a message to $S1$ requesting resource $R1$, and gets blocked. At this point the resource controller at site $S1$ reports to DD indicating $P1$ to be in possession of $R1$, and $P2$ to be in wait for it.

Next, $P1$ releases $R1$ and requests $R2$. The corresponding request message, arriving at $S2$, causes $P2$ to get blocked. The resource controller at $S2$ now reports to DD indicating $P2$ to be in possession of $R2$ and $P1$ to be waiting for it. On putting these two reports together, DD detects a cyclic wait $P2 \rightarrow R1 \rightarrow P1 \rightarrow R2 \rightarrow P2$ and may detect a deadlock unless its algorithm takes other steps to verify that a cycle really exists. In this case the cycle does not exist, since $P1$ is no longer in possession of $R1$.

Another danger is that a deadlock detection algorithm may fail to detect a deadlock that really exists. Typically this happens when an algorithm fails to take into account that messages may arrive after indefinite delays. As a result, all the deadlock computations that arise as a result of a sequence of events causing a deadlock, may operate on incomplete information, and thus the deadlock goes undetected.

4.4. Terminology and Assumptions

The database is accessed and updated through *transactions*. A transaction consists of one or more processes, called *agents*. An agent may request to acquire either of two kinds of resources: *reusable resources* (which will be referred to simply as *resources* from now on) and *consumable resources* (

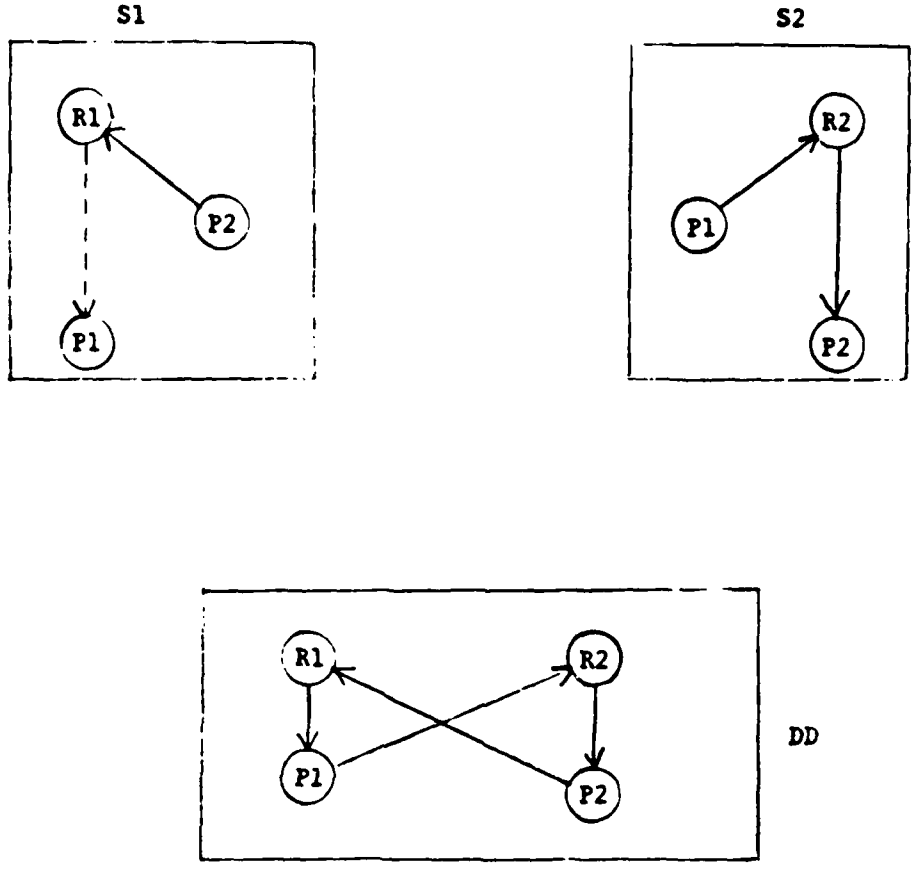


FIG. 4.1. RACE CONDITIONS IN DEADLOCK DETECTION

which will be referred to as *messages* from now on). Messages are used by agents of a transaction to co-ordinate their work. In the case of resources, it is assumed that any agents currently in possession of a resource must *all* relinquish it before *any* of the agents currently waiting for the resource can gain possession of it. This assumption is necessary to make existence of a cycle a sufficient condition for deadlock.

A transaction agent may be in one of two states: *active* or *waiting*. Initially it is in *active* state. It may enter *waiting* state if:

(i) it wishes to receive messages from each of a set of agents belonging to the same transaction. For example, the agent co-ordinating the commit processing for the transaction may enter *waiting* state and remain there till it has received *prepared-to-commit* messages from all other agents of the transaction.

(ii) it wishes to acquire each of a set of resources (in specific modes (e.g. *shared*,*exclusive*)).

When *all* the messages or resources have been received, the agent reenters *active* state.

4.5. Centralized Algorithms

Early work in centralized deadlock detection [GRA 78, GOL 77] does not correctly solve the problem of race conditions. The algorithm of [GRA 78] works under the assumption of *two-phase* usage of resources (explained below). However, if this assumption is made, an algorithm which is much more efficient can be constructed as shown later. Similarly, a timing problem in the centralized algorithm of [GOL 77] was shown in [SUN 78].

In [HO 79] two algorithms were proposed to address the problem of race conditions. The first is a two-phase algorithm in which first one set of reports

is collected from all sites and then another set is collected. Only the information common to the two sets of reports is assembled to check for global deadlocks and it is shown that spurious indications are thereby avoided. In the second algorithm, each inter-site arc is replicated at both the sites involved and a deadlock is detected after only one set of reports is received.

However, these algorithms require that all sites in the network which access the resources as well as the sites controlling the resources should report to the deadlock detector. Typically, the number of sites controlling the resources will be much smaller than the number of sites accessing the resources. For example, in a network running distributed INGRES [STO 79], the control is done only from the *primary* sites in the network. In the next two sections, we show under what conditions we can construct algorithms which utilize reports from only the resource-controlling sites.

4.5.1. Detection under Conditions of 2-Phase Resource Usage

By 2-phase usage [ESW 76] of resources, we mean that the execution of a transaction can be divided into two distinct phases: a *growing* phase and a *shrinking* phase, the latter following the former. In the *growing* phase, resources are acquired but not released. In the *shrinking* phase, resources are released but not acquired. The implication is that, under this discipline, a transaction does not release any resources until after it has acquired all the resources it needs.

Suppose that only the resource controllers send reports to the deadlock detector, giving for each resource the list of transactions in possession, and the list of transactions in wait. The identity of the specific agent of the transaction which is in possession or waiting is not given. Periodically, the deadlock detector takes the latest report from each controller and

assembles the reports.

Thm 4.1: Suppose a cycle is found in the transaction-resource graph created by the above procedure. Then the transactions in the cycle are deadlocked.

Proof: Let the cycle take the form shown in Fig. 4.2.

For each resource node in the graph, the incoming arcs represent transaction agents waiting to acquire the resource and the arcs running out of the node transactions in possession of the resource. By our earlier assumption that all requested resources must be acquired before the agent making the request can proceed, none of the arcs representing a wait for the resource can vanish before *all* the arcs representing possession of the resource vanish.

For each transaction node, the incoming arcs represent resources in possession and the outgoing arcs represent resources the transaction is waiting for. Since each transaction uses resources in a 2-phase manner, no incoming arc at a transaction node can vanish before all its outgoing arcs vanish.

Let $x<y$ indicate that arc y can vanish only after arc x vanishes. Applying the arguments given above to the incoming and outgoing arcs at the nodes $R1, T2, R2, T2 \ldots\ldots, TN, RN$ we get $a_1<a_2<a_3<\ldots.<a_{2N-2}<a_{2N-1}<a_{2N}$ i.e. $a_1<a_{2N}$. But applying them to the incoming and outgoing arcs at node $T1$, it follows that $a_{2N}<a_1$. This contradiction implies that no arc in the cycle can vanish (unless one of the transactions in the cycle is aborted). Hence the cycle represents a genuine deadlock.

—

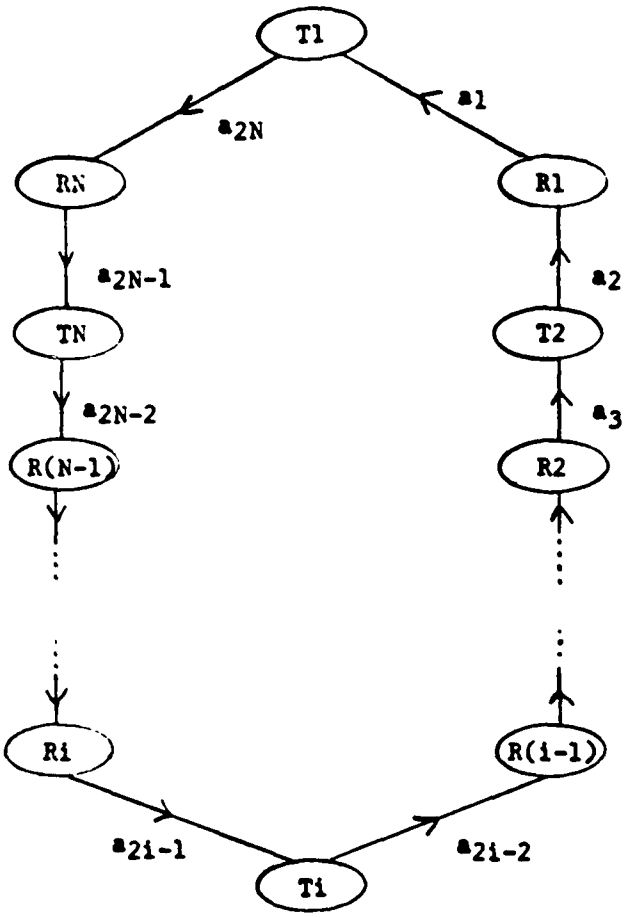


FIG. 4.2. CYCLE IN PROOF OF THEOREM 4.1.

Further, every genuine deadlock will be detected, since none of the arcs in the deadlock cycle will vanish till the deadlock is broken and thus the cycle must eventually appear in the deadlock detector's assembled graph.

Thus, we have shown that when resource usage is 2-phase, status reports need be collected only from the resource-controllers. All algorithms for centralized detection published hitherto have involved gathering reports from all sites in the network. Two-phase usage of resources is used in many systems to satisfy the requirement of *serializability* of transaction execution histories [ESW 76]. Therefore the detection procedure described above can be utilized in these systems e.g. distributed *INGRES*.

4.5.2. Detection under Conditions of non-2-phase Resource Usage

In some database systems, resource usage is not constrained to be necessarily 2-phase. For example, System R [AST 76] allows three different degrees of data consistency from which the user may specify one for his transaction. The highest degree of consistency is the one corresponding to 2-phase resource usage. The advantage of using lower degrees of consistency is less lock contention.

When the resource usage is not necessarily 2-phase, the possibility of false indication of deadlock in situations such as the one illustrated in Fig. 4.1. arises. For the most general case, where the agents of a transaction may execute in parallel, an algorithm such as those described in [HO 79, GRA 78], in which all sites in the network have to send status reports to the deadlock detector, is necessary. However, there is one transaction model in which intra-transaction concurrency is not present. This is the "migrating" transaction model [ROS 77, GRA 81], used in System R^*, the distributed version of System R. Here, a transaction starts at one site and moves from site

to site as necessary to access remote resources. At every site visited by the transaction, there is a single agent that does the work at that site on behalf of that transaction. The agent of the transaction at the site that the transaction is currently visiting is called the *front* of the transaction. A list of unreleased resources is carried along by the transaction front and messages are sent releasing them when the transaction terminates. (Acquired resources may be released prior to transaction termination, for example if the highest degree of data consistency is not desired for the transaction.) It is assumed that the transaction front does not migrate from a site before it has acquired the resources it has requested while at that site.

A global clock facility fulfilling Lampson's clock rules [LAM 78a], mentioned in Section 2.2.2 of Chapter 2, is assumed to exist. Timestamps are assigned to resource requests using this facility. Uniqueness of timestamps is assured by taking the clock reading to include the site id as its less significant part. These rules imply that

(i) given two requests issued by a transaction front while at a given site, the timestamp assigned to the later request is greater than that assigned to the earlier request, and

(ii) if the transaction front migrates from site a to site b, then timestamps associated with requests issued at b are greater than those associated with requests issued by the front when at a.

Each request for a resource sent to a resource controller by a transaction front is accompanied by the timestamp assigned to the request. This timestamp is retained by the controller till it is informed about the release of the resource by the transaction. The resource may be released by the front at a site other than the one where it was requested and acquired.

Further, as mentioned before, the transaction front maintains a list of resources which it has acquired but not released and the corresponding request timestamps.

From time to time, a site may receive a *confirm_ownership* message from the deadlock detector. This message specifies a transaction T, a resource R and a timestamp t. The site returns a positive acknowledgement if

(i) the front of the transaction T is currently at the site and

(ii) in the list of unreleased resources maintained by the transaction front, the resource R is present with associated timestamp t.

The site returns a negative acknowledgement otherwise.

Periodically, every resource controller sends a report to the deadlock detector, giving for each resource under its control

(i) the set of transactions in possession, along with the timestamps of the corresponding requests and

(ii) the set of transactions waiting, along with the timestamps of the corresponding requests.

The deadlock detector executes the following algorithm:

(i) Periodically, it selects the latest report from each resource controller and assembles the reports.

(ii) If one or more cycles is detected in the assembled graph, the following procedure is executed for each cycle C:

Let C be the sequence of arcs $T0 \rightarrow R0 \rightarrow T1 \rightarrow R1 \ldots\ldots \rightarrow T(N-1) \rightarrow R(N-1) \rightarrow T0$.

Let $t_w(i)$, $i=0,\ldots N-1$, be the timestamp associated with the arc $Ti \rightarrow Ri$.

Let $t_e(i)$, $i=1,...N-1$, be the timestamp associated with the arc $R(i-1)\rightarrow Ti$ and let $t_e(0)$ be the timestamp associated with the arc $R(N-1)\rightarrow T0$.

If for every transaction Ti, $i=0,1,..N-1$, in cycle C, $t_e(i)\leq t_w(i)$ then

(a) to every site ORIG($t_w(i)$),$i=1,...N-1$, send a *confirm_ownership* message (Ti,$R(i-1)$,$t_w(i)$) [ORIG(t) represents the site at which timestamp t is issued, and can be computed from t itself.] and to ORIG($t_w(0)$) send a *confirm_ownership* message ($T0$,$R(N-1)$,$t_w(0)$).

(b) if all acknowledgements are positive, declare cycle C to represent a deadlock.

Thm 4.2: Every cycle C declared to represent a deadlock represents a true deadlock.

Proof:The argument is the similar to that for Thm 4.1, namely that for each node in the cycle, the incoming arc can vanish only after the outgoing arc. This holds for resource nodes for the same reason as before. It holds for a transaction node Ti because

(i) since $t_e(i)\leq t_w(i)$, the resource request for the acquired resource occurred at the same time or before the resource requested by Ti and

(ii) at a time later than $t_w(i)$, the acquired resource has not been released, since a positive acknowledgement from ORIG($t_w(i)$) is received.

Therefore C represents a genuine deadlock.

—

Thm 4.3: Every genuine deadlock is detected.

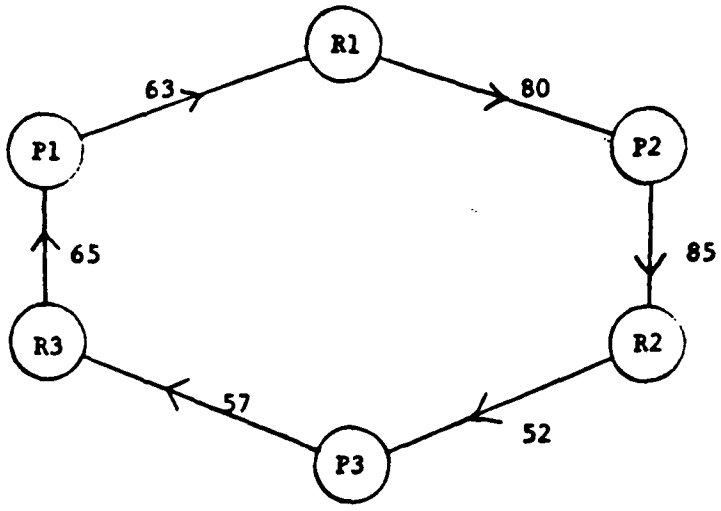
Proof: Let the deadlock be represented by the cycle

$C: T0 \rightarrow R0 \rightarrow T1 \ldots \rightarrow T(N-1) \rightarrow R(N-1) \rightarrow T0$ with $t_e(i)$, $t_w(i)$, $i=0,1 \ldots N-1$ being defined as before. Since the deadlock is genuine, none of its arcs will vanish until the deadlock is broken. Hence the cycle will be detected by the deadlock detector. Further, since the resource acquired by Ti, $i=0,..N-1$, in the cycle C must have been requested at the same time or before it requests Ri, $t_e(i) \leq t_w(i)$. Therefore, the deadlock detector will send out *confirm-ownership* messages to ORIG($t_w(i)$), $i=0,...N-1$. Since the deadlock is genuine, the front of transaction Ti will be trapped at the site it requested Ri, i.e. ORIG($t_w(i)$). Hence positive acknowledgements will be received for all the *confirm_ownership* messages sent out and a deadlock will be declared.

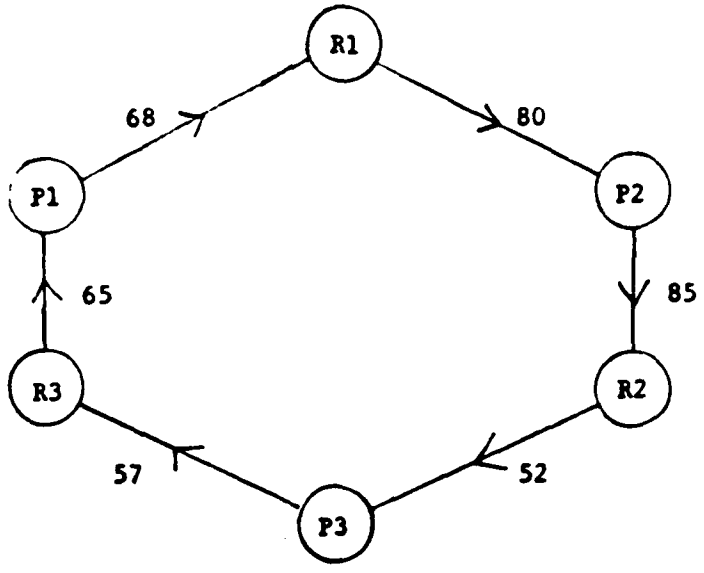
Since a cycle is likely to occur only rarely in the assembled graph at the deadlock detector, the *confirm_ownership* messages will occur only rarely. Hence, the participation of non-resource-controlling sites will be only rarely required for deadlock detection. The lower communication overhead that this algorithm causes is obtained at the expense of a larger time to detect a deadlock compared to the one-phase algorithm in [HO 79]. The *confirm-ownership* messages and acknowledgements constitute an extra phase which increases the detection time by one round-trip delay.

Fig. 4.3.a shows a case where the extra phase is not initiated since $t_e(1) > t_w(1)$. Fig. 4.3.b shows a case which does invoke the extra phase.

4.6. Distributed Detection



(a)



(b)

FIG. 4.3. EXAMPLES OF CYCLES WHICH DO NOT, AND DO INITIATE CONFIRM-OWNERSHIP MESSAGES RESPECTIVELY

4.6.1. Past Work

The first distributed detection algorithms are in [CHA 74, MAH 76]. In both, issued requests for resources are divided into those that are incapable of causing a global deadlock and those that are capable of doing so. In the latter case, resource tables from all sites in the network are assembled to check if a global deadlock exists. Besides causing excessive communication overhead, both algorithms have been shown in [GOL 77] to fail in detecting certain kinds of deadlocks. In the "on-line" algorithm of [ISL 78], a complete global view of resource status is maintained at each site. This algorithm also suffers from excessive communication overhead.

[GOL 77] presents a distributed algorithm which is similar to many subsequently-appearing algorithms [MEN 79, CHA 82, OBE 82, BAD 83]. The common element in these algorithms is *forward* traversal of the global status graph (i.e. traversal in the direction of the graph arcs), which may cause the deadlock computation to migrate from site to site as intersite arcs are encountered. In [BAD 83], a request from a transaction is accompanied by its previous lock history; this hastens detection of intersite deadlock cycles of length two. In [OBE 82], an intersite arc is traversed only if the tail node id is greater than the head node id; this optimization reduces the number of deadlock detection messages caused by a cycle of arcs by half. In [MEN 79], the results of graph traversals are also recorded in the graph in the form of arcs representing indirect independencies (i.e. a chain of arcs $a1 \rightarrow a2 \rightarrow .... \rightarrow aN$ may cause the addition of an arc $a1 \rightarrow aN$). However, there is no provision in the algorithm for updating this "condensed" information, and hence false deadlocks may be detected. It was shown in [GLI 80] that the algorithm also fails to detect *some* deadlocks. One of the authors of

[MEN 79] proposed a solution, presented in [GLI 80], purporting to remedy this last deficiency. But in [TSA 82] it was shown that this solution, too, does not detect all deadlocks.

[TSA 82] proposes a solution which differs from previous solutions in that it traverses the graph in reverse, i.e. it proceeds from one transaction to another waiting for the first to release some resource. With forward traversal of a chain of arcs, the transactions in the chain yet to be traversed must release one or more resources before the transactions in the chain already traversed can leave their *waiting* state. But with reverse traversal, this is not true. Therefore, if a node already encountered is re-encountered in the backward traversal, the algorithm must verify that the forward path to that node still exists before declaring a deadlock. In fact, [TSA 82] indicates deadlock in some cases where they do not exist.

In Fig. 4.4.a, a chain of arcs $R20 \rightarrow T21 \rightarrow R21 \rightarrow T22 \rightarrow R22 \ldots \rightarrow T2i \rightarrow R2i \rightarrow T2(i+1)$ is shown. Suppose the request of $T2i$ for $R2i$ occurs at t_0. In the algorithm of [TSA 82], the wait of $T2i$ for $R2i$ will be propagated backwards in the form of a "reaching edge". This, when it reaches the site where $T21$ resides, creates the "resource reaching edge" $T21 \rightarrow R2i$.

In Fig. 4.4.b, transaction $T2(i+1)$ has released $R2i$ which is now free.

In Fig. 4.4.c, transaction Tx acquires $R2i$. Let this be at time t_1. Then it requests resource $R10$ which is the first node in the chain $R10 \rightarrow T11 \rightarrow R11 \rightarrow T12 \rightarrow \ldots \rightarrow R1(j-1) \rightarrow T1j$.

Next, in Fig. 4.4.d, $T1j$ requests the resource $R20$ and is blocked. Let this occur at $t_2 > t_1$. The indirect wait of $T21$ for $R2i$ (which is, unknown to the site where $T21$ resides, not valid any more) is propagated backwards till

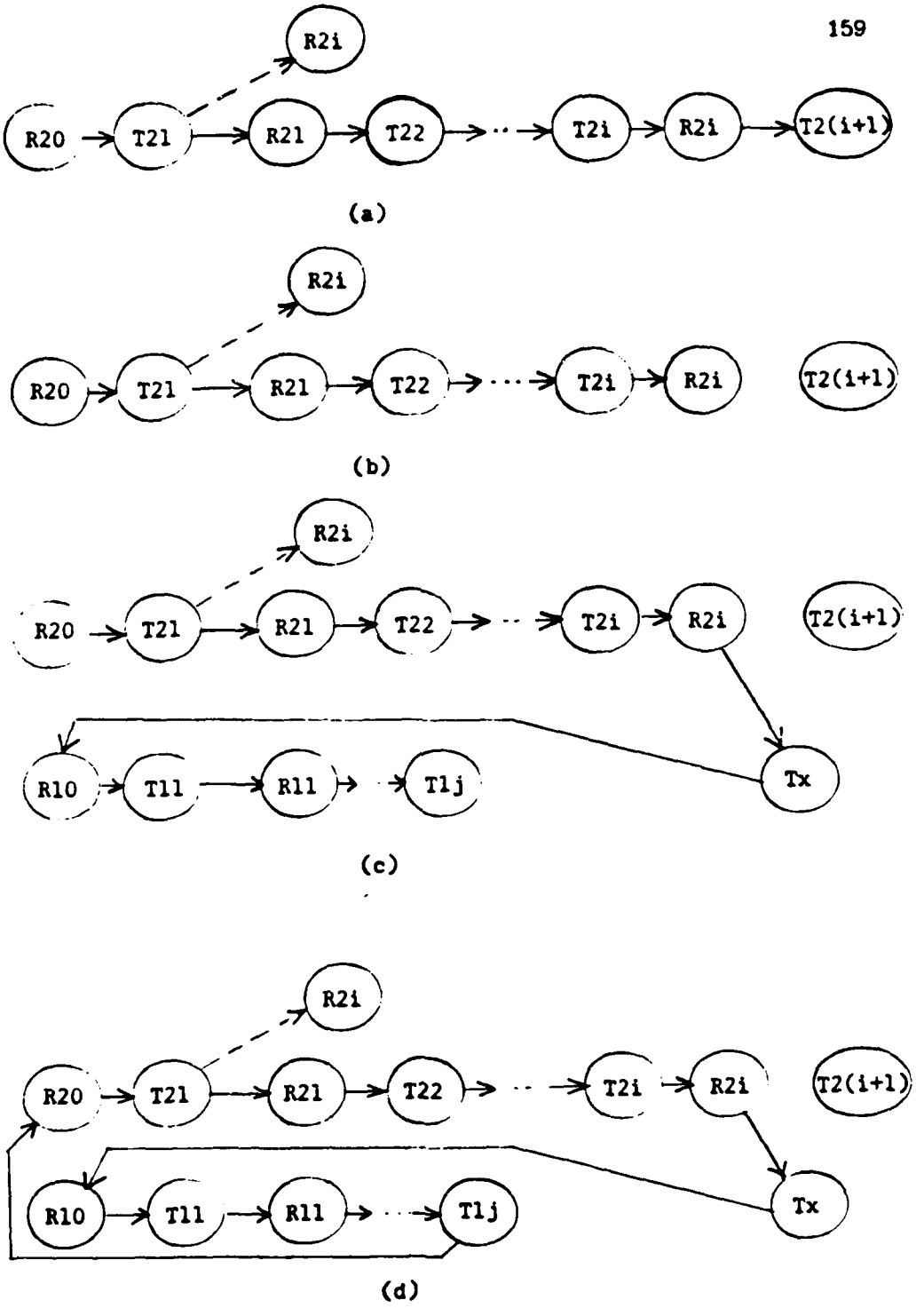


FIG. 4.4. COUNTEREXAMPLE TO ALGORITHM IN [TSA 82].

the resource reaching edge $Tx \rightarrow R2i$ is created, forming a cycle. Moreover, the timestamp that accompanies the reaching edge is t_2, which will indicate that some transaction reachable by following arcs forward from Tx requested $R2i$ after Tx acquired it. Hence, the algorithm declares deadlock. Thus, although the timestamp mechanism was introduced to prevent spurious indications of deadlock, a false deadlock is detected here.

It does not appear difficult to correct this error. One solution appears to be to associate a timestamp of t_0 with every resource reaching edge whose creation originated with a resource request occurring at t_0. However, there is another reason why the algorithms of [TSA 82, OBE 82, CHA 82, BAD 83] are deficient in comparison with centralized algorithms. In a centralized algorithm, a deadlock can be detected by the detector site one message delay after the last arc of the deadlock cycle comes into existence. But with the distributed algorithms just mentioned a delay equal to the time to go around the cycle is usually required. If all arcs of the cycle come into existence more or less simultaneously this delay cannot be avoided. But if the last arc comes into existence an appreciable time after all or most of the other arcs come into existence, it should be possible to reduce the detection time by the use of "condensed" information. In the algorithm proposed below, both forward and backward traversal are used to achieve this goal. Further, a timestamp mechanism is used to prevent false deadlock indications. In this algorithm, timestamps have no ordering role to play, but act as unique identifiers.

4.6.2. A Distributed Detection Algorithm

4.6.2.1. Terminology

In the algorithm proposed below, the information necessary to detect deadlocks is represented in the form of a transaction_agent-resource-message (*TRM*) graph at each site. In the graph, there are *agent*, *resource* and *message* nodes. The former two types of nodes represent transaction agents and resources respectively. $T.x$ represents the agent of transaction T at node x. For each pair of communicating agents $T.a$ and $T.b$ of a transaction T at sites a and b respectively, there can exist two nodes $M(T,a,b)$ and $M(T,b,a)$, the former representing messages sent by $T.a$ to $T.b$ and the latter representing messages sent by $T.b$ to $T.a$. Let $TRM(s)$ be the *TRM* graph at site s. Then the nodes in $TRM(s)$ representing (a) transaction agents residing at site s (b) resources whose lock controllers are at site s and (c) messages sent to a transaction agent local to site s are said to be *local* nodes. Other nodes are *non-local*.

Fig. 4.5 shows an example involving four sites a,b,c, and d. Local to site a are the agent nodes $T1.a$,$T2.a$; the resource node $R1$ and the message nodes $M(T1,b,a)$ and $M(T1,c,a)$. Local to site b are the agent nodes $T1.b$, $T6.b$, the resource node $R2$ and the message node $M(T1,a,b)$. Local to site c are the agent nodes $T1.c$, $T3.c$, the resource node $R3$ and the message node $M(T1,a,c)$. Local to site d are the agent nodes $T4.d$, $T5.d$, the resource nodes $R4$, $R5$, $R6$.

The transaction $T1$ is a distributed transaction. $T1.a$ does the commit co-ordination of the transaction; it also updates $R1$. As can be seen from the figure, $R1$ has been locked by $T1.a$. It is not waiting for $M(T1,b,a)$ since $T1.b$ has already sent the *prepared-to-commit* message to $T1.a$. $T1.b$ is supposed to update $R2$ on which it has obtained a lock. But $T1.c$, which is

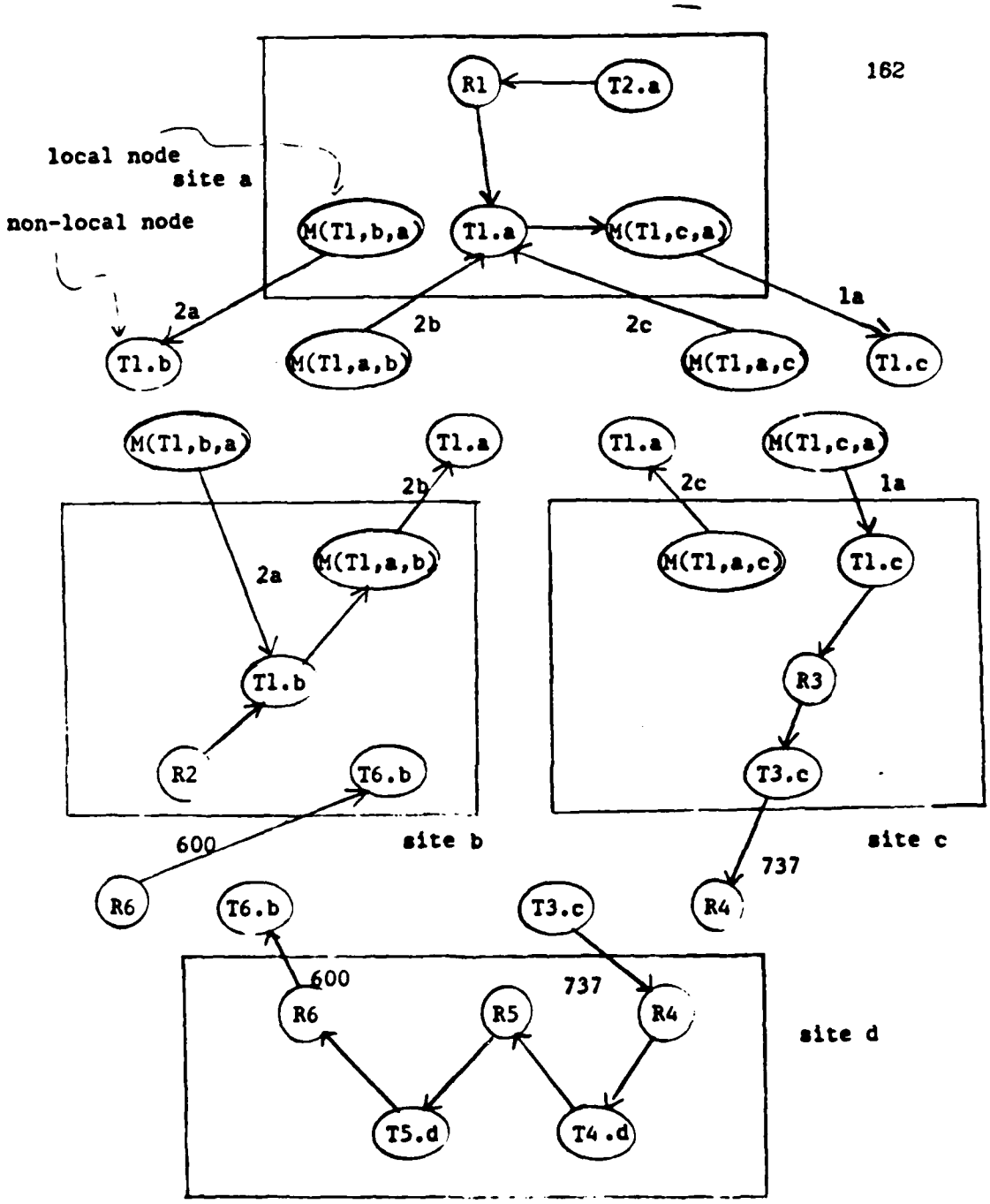


FIG. 4.5. EXAMPLE TO ILLUSTRATE TYPES OF NODES AND ARCS USED IN THE DISTRIBUTED ALGORITHM

supposed to update $R3$, has not yet got a lock on it and hence it has not yet returned a *prepared−to−commit* message to $T1.a$, which is therefore waiting for $M(T1,c,a)$. $T1.a$ has sent one message each to $T1.b$ and $T1.c$ (conveying the work they are to do), hence the arcs from $M(T1,a,b)$ and $M(T1,a,c)$ to $T1.a$ are marked $2b$ and $2c$ respectively , signifying that the second message from $T1.a$ to $T1.b$ and the second message from $T1.a$ to $T1.c$ are in $T1.a$'s possession, i.e. have not yet been sent by it. The second part of the marking refers to the site to which the message is being sent. Since $T1.b$ has sent one message to $T1.a$, the arc from $M(T1,b,a)$ to $T1.b$ is marked $2a$. The transaction $T2.a$ is waiting to lock $R1$.

At site c, $T1.c$ has not yet sent its *prepared−to−commit* message to $T1.a$, hence the marking $1a$ on the arc from $M(T1,c,a)$ to $T1.c$. $T1.c$ is waiting to lock $R3$, which has been locked by $T3.c$. $T3.c$ is waiting to lock $R4$ whose lock controller is at site d. The number 737 on the corresponding arc is a timestamp showing the time at which the lock request was made.

At site d, $T4.d$ has locked $R4$ and is waiting for a lock on $R5$ which has been locked by $T5.d$. $T5.d$ in turn is waiting to lock $R6$ which has been locked by the $T6.b$. The marking 600 on the arc from $R6$ to $T6.b$ is the time at which the lock was granted.

Note that arcs between nodes local to two different sites are reproduced at both sites, and that they have a marking corresponding to a message number concatenated with a site id, or a timestamp. This marking is done so that the graphs at different sites can be put together consistently for graph traversals. Arcs between nodes local to the same site are not marked and are called *internal* arcs.

At any site, arcs running from non-local nodes to local nodes are referred to as *direct incoming arcs (DIAs)* and the corresponding local nodes are referred to as *in-nodes*. Arcs running from local nodes to non-local nodes are referred as *direct outgoing arcs (DOAs)* and the corresponding local nodes are referred to as *out-nodes*. At site a in Fig. 4.5, $T1.a$ is an in-node and $M(T1,b,a)$ and $M(T1,c,a)$ are out-nodes. The arcs $M(T1,a,b) \rightarrow T1.a$ and $M(T1,a,c) \rightarrow T1.a$ are DIAs and the arcs $M(T1,c,a) \rightarrow T1.c$ and $M(T1,b,a) \rightarrow T1.b$ are DOAs. Note that a DIA at one site is a DOA at another, and vice versa.

The outgoing arcs defined so far represent direct relationships between out-nodes at one site and in-nodes at another. In order to speed up deadlock detection, condensed information in the form of *indirect outgoing arcs (IOAs)* are maintained. IOAs represent indirect relationships between out-nodes at one site and in-nodes at a different or the same site.

Fig. 4.6 shows Fig.4.5 augmented with IOAs shown in dotted arcs. For example, the out-node $M(T1,a,b)$ at site b and the in-node $T6.b$ at the same site have a sequence of arcs connecting them: $M(T1,a,b) \rightarrow_{2b} T1.a$, $T1.a \rightarrow M(T1,c,a)$, $M(T1,c,a) \rightarrow_{1a} T1.c$, $T1.c \rightarrow R3$, $R3 \rightarrow T3.c$, $T3.c \rightarrow_{737} R4$, $R4 \rightarrow T4.d$, $T4.d \rightarrow R5$, $R5 \rightarrow T5.d$, $T5.d \rightarrow R6$, $R6 \rightarrow_{600} T6.b$. This connecting sequence of arcs is represented concisely by the IOA $M(T1,a,b) \rightarrow_{600} T6.b$ at site b. Each IOA is *associated* with a DOA, namely the DOA which is first in the connecting sequence. Thus the IOA $M(T1,a,b) \rightarrow_{600} T6.b$ is associated with the DOA $M(T1,a,b) \rightarrow_{2b} T1.a$. If $T6.b$ tries to lock $R2$ in a mode incompatible with the mode in which $T1.b$ has locked it, the arc $T6.b \rightarrow R2$ will be created causing a deadlock to be detected

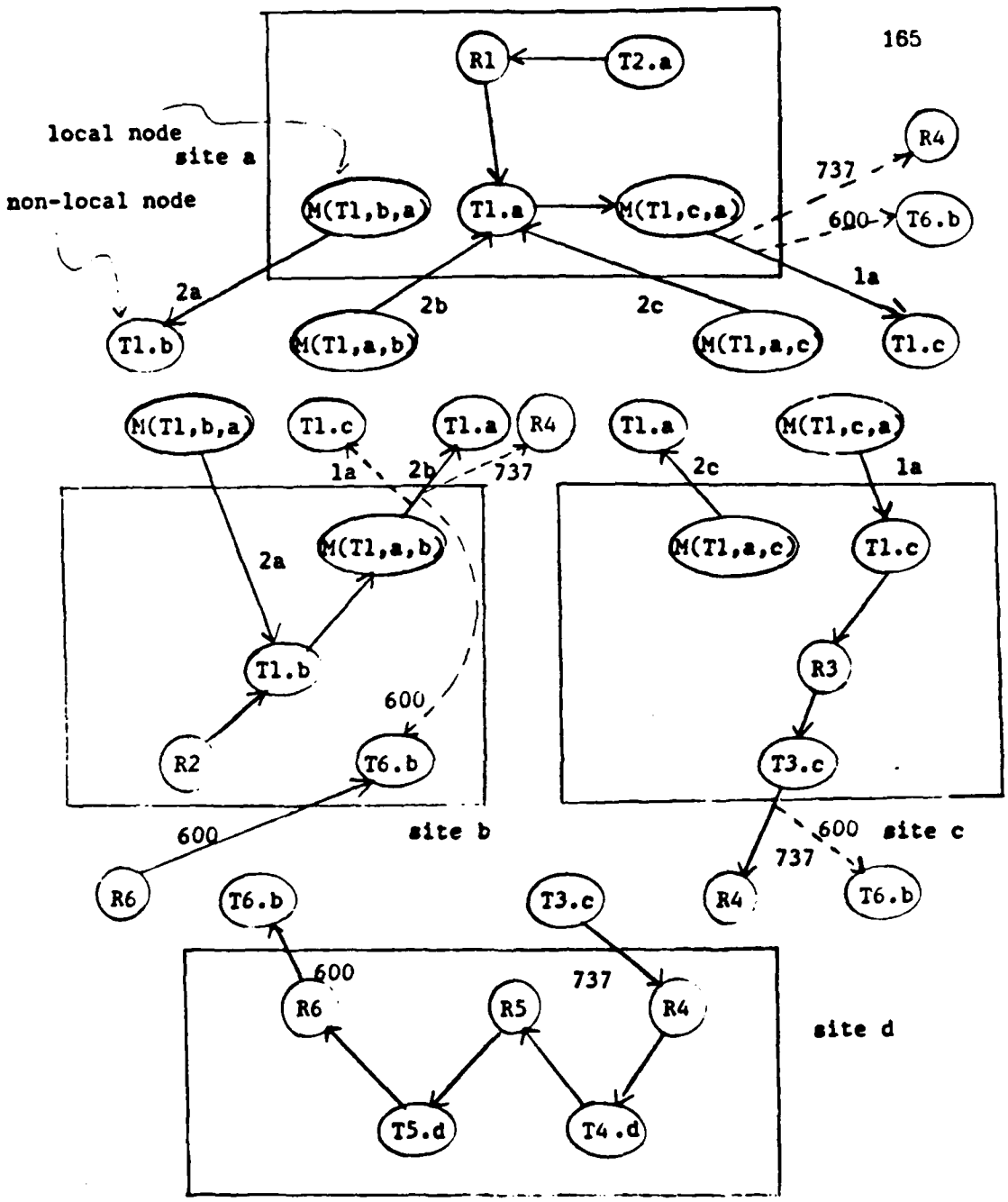


FIG. 4.6. THE PREVIOUS FIGURE WITH DOAs INCLUDED.

(shown in dotted lines)

at site b due to the cycle $T1.b \rightarrow M(T1,a,b) \rightarrow_{600} T6.b \rightarrow R2 \rightarrow T1.b$. Thus the deadlock is detected at once instead of having to wait for several message delays till information from all four nodes is gathered as in, e.g., [OBE 82].

If an arc a runs from node x to node y, we refer to x as the *head* and to y as the *tail* of a, respectively. DIAs and DOAs have message numbers concatenated with site ids or timestamps associated with them, which are called the *marks* of these arcs. The *arc-identifier* of a DIA or DOA, d, is the pair of values (head(d),mark(d)). The IOAs associated with a DOA are stored as arc-identifiers in the *ioas* field of the DOA.

The algorithm utilizes timeout periods in such a manner that under lightly loaded conditions, i.e., when requested locks and messages become available to the requestor within the specified timeout periods, and acquired locks are released within specified timeout periods, no checks for the existence of deadlock cycles or attempts to construct IOAs in order to hasten the detection of deadlocks occur. For this reason, with each arc there is an associated field *timed_out* taking the values *TRUE* or *FALSE* according as the timeout period for the arc has completed or not. [In some cases, as will be seen, there will be no need to even start the timeout, so the *timed_out* field takes the value *TRUE* as soon as the arc is created.]

Not only a wait by an agent for a resource or message, but also the granting of a resource to a message may cause a deadlock [ISL 80]. If, however, the grant causes the transaction agent to enter *active* state, no deadlock can occur as a result of such a resource grant and the algorithm can be optimized accordingly.

In the next section, we present the algorithm for deadlock detection as executed at a site s. In order to distinguish units of communication

exchanged between between transaction agents and those between resource controllers for the purpose of resource allocation and deallocation as well as deadlock detection, we refer to the latter as *signals*, the former being referred to, as before, as messages.

There are 5 kinds of signals:

(i) *resource request* (*RR*): This signal is sent when a lock on a non-local resource is requested by a local transaction agent. The signal carries information identifying the requesting agent, the name of the resource and the mode of lock desired, and is accompanied by a timestamp TS_R generated at the requesting site.

(ii)*resource grant* (*RG*): This signal is sent in response to a RR signal, to the requesting site. In addition to information identifying the RR signal to which it is a response, it carries a timestamp TS_G generated at the site that is sending the RG signal.

(iii) *resource free* (*RF*): This signal is sent to the resource controller at another site when a local transaction agent no longer requires a lock on a remote resource under the control of that resource controller. It carries information that enables the resource controller to delete the appropriate DOA and associated IOAs.

(iv) *agent create* (*AC*): When a local agent $T.s$ wishes to create an agent at another site r, this signal is sent to the site r. A full duplex channel is established between the two agents.

(v) *backward propagation* (*BP*): This signal is used to establish IOAs to speed up deadlock detection. It has two fields : *IA_SET* and *OA_SET*. The former is

a set of DIAs, with the tail of each DIA being local to the same site r, the site to which the BP signal is sent. OA_SET is a set of arc-identifiers, corresponding to a subset of the DOAs at site s and associated IOAs. On receipt of a BP signal, further BP signals may be sent. The BP signals flow backwards along the TRM graphs.

(vi) *forward propagation (FP)*: This signal is sent in order to detect possible multisite deadlocks. Each FP signal may spark off further FP signals at the recipient site, and these signals are said to belong to the same *deadlock computation*. Basically, the FP signals of a deadlock computation traverse the TRM graphs in the forward direction. A FP signal has five fields:

(a) ORIGIN: This is the id of the site that began the deadlock computation to which the FP signal belongs.

(b) VICTIM: This is the transaction that is to be aborted if the deadlock computation finds one or more deadlock cycles.

(c) CHECK_SET: This is a set of DIAs at the ORIGIN site. If the deadlock computation reaches one or more DOAs or IOAs at some site corresponding to one or more members of the CHECK_SET, then the transaction VICTIM is deadlocked and must be aborted.

(d) TRAVERSED_SET: This is a set of arc-identifiers corresponding to DIAs along which the deadlock computation has already traveled, hence no new traversals along these DIAs should be initiated in this deadlock computation.

(e) OA_SET: This is a set of arc-identifiers corresponding to DIAs at the site to which the FP signal is being sent. These DIAs are the arcs along which the graph traversal is continued at the recipient site.

The resource controller algorithm is event-driven. The relevant events are

(i) timeouts and arrivals of signals

(ii) locally originating requests for (a) obtaining locks on resources and releasing them, (b) creating agents at other sites and (c) sending to, and waiting to receive messages from, agents of the same transaction at other sites.

It is assumed in the following description that when a transaction completes, the nodes representing its agents and arcs incident at them are deleted. C_L represents the current value of the local clock.

4.6.2.2. Description of Algorithm

Below we describe the actions taken by the resource controller on the occurrence of each event:

Requesting, Granting and Freeing Resources

(1) Agent *T.s* requests resources *R*1, *R*2, *R*3....,*RN* (in specified modes)

(a) If not all requested resources are local, or if not all requested local resources are available in the required modes, set the status of *T.s* to *waiting*.

(b) For each resource available in the required mode, create the appropriate internal arc (indicating resource possession), with *timed−out* set to *TRUE*. (COMMENT: A timeout need not be started for this arc, since if its creation results in a deadlock cycle, it will be detected when the timeouts for the arcs created in (c), (d) below, complete.)

(c) For each local resource unavailable in the required mode, create

the appropriate internal arc (indicating resource wait), with *timed_out* set to *FALSE*. Start a timeout of period $T1$.

(d) For each non-local resource, create the appropriate DOA (indicating resource wait), with *timed_out* set to *FALSE*, *mark* set to C_L, and *ioas* set to null. Start a timeout of period $T2$ and send a RR signal to the site that controls the resource, with TS_R, set to the same value as the *mark* field.

(2) A RR signal arrives from $T.s'$, $s' \neq s$ for resource Ri with a timestamp TS_R.

(a) If the resource is available in the required mode,

(i) create the appropriate DOA (indicating resource possession), with *timed_out* set to *FALSE*, *mark* set to C_L and *ioas* set to null, and start a timeout of period $T4$.

(ii) send a RG signal to the requesting site, with TS_C set to the same value as the *mark* field.

(b) If the resource is unavailable in the required mode, create the appropriate DIA (indicating resource wait), with *timed_out* set to *FALSE*, *mark* set to TS_R and start a timeout of period $T1$.

3. A RG signal in response to a request from a local agent $T.s$ for a non local resource R arrives, with timestamp TS_C.

(a) Delete the DOA that represents $T.s$ waiting for R and abort the associated timeout if *timed_out* is *FALSE*.

(b) Create the appropriate DIA (indicating resource possession) setting the *mark* field to TS_C. If no outgoing arcs remain at the node $T.s$, set its status to *active* and set *timed_out* on the DIA to *TRUE*. Otherwise set the field to *FALSE* and start a timeout of period $T3$.

(4). Agent T.s releases resources $R1, R2, R3 ... RM$.

(a) For each Ri that is local,

(i) delete the appropriate internal arc (indicating resource possession), aborting the associated timeout if *timed_out* is *FALSE*.

(ii) choose, if possible, a set of transaction agents waiting to access Ri that can now be given access to it.

(iii) for these agents, delete the appropriate internal arcs or DIAs (indicating resource wait), aborting associated timeouts when the *timed_out* fields are *FALSE*.

(iv) for each non-local agent, follow the steps given in 2(a)(i) and 2(a)(ii).

(v) for each local agent granted access to Ri in (ii) above, create the appropriate internal arc (indicating resource possession). If no outgoing arcs remain from the agent node, set its status to *ready* and set the *timed_out* field on the internal arc to *TRUE*. Otherwise, set the field to *FALSE* and start a timeout of period $T3$.

(b) For each non-local resource, delete the appropriate DIA, aborting the associated timeout if *timed_out* is *FALSE*. Send a *RF* signal to the site controlling the resource.

(5) A RF signal, indicating that $T.s'$ has released a resource Ri local to s, arrives.

(a) Delete the DOA from Ri to $T.s'$, aborting the associated timeout if *timed_out* is *FALSE*.

(b) Perform steps 4(a)(ii)-(v).

Agent Creation, Sending and Receiving Messages

(6) $T.s$ requests creation of an agent at site $r \neq s$.

(a) Create a DIA (indicating message-possession) from $M(T,s,r)$ to $T.s$ with *mark* set to $1r$ and *timed_out* set to *TRUE*.

(b) Create a DOA (indicating message possession) from $M(T,r,s)$ to $T.r$ with *mark* set to $1s$ and *timed_out* set to *TRUE*. COMMENT: Note that creation of DIAs and DOAs that indicate message possession do not create a cycle and hence no timeout need be started. It is *waiting* to receive a message that can complete a cycle.

(c) Send a AC signal to site r.

(7) A AC signal is received is received from site $r \neq s$ for creation of an agent of transaction T'.

Carry out steps 6(a) and 6(b) with T' replacing T.

(8) $T.s$ sends a message to $T.r1, T.r2, T.r3 ... T.rK$.

For each rj,

(a) increment the message number in the *mark* field on the DIA from $M(T,s,rj)$ to $T.s$ by 1.

(b) send the message to site rj.

(9) $T.s$ waits for a message from $T.r1, T.r2 .... T.rL$.

(a) If at least one message is queued for $T.s$ from each of $T.r1, T.r2 ... T.rL$, then remove the message at the head of each queue

and supply the messages to $T.s$.

(b) If at least one of the queues is empty

(i)set the status of $T.s$ to *waiting*.

(ii) For each ri such that there are no messages queued from $T.ri$ to $T.s$, create an internal arc (indicating message wait) from $T.s$ to $M(T,ri,s)$ with *timed_out* set *FALSE* and start a timeout of period $T5$.

(iii) For each ri such that the queue of messages from $T.ri$ to $T.s$ is non-empty, remove the message at the head of the queue and supply it to $T.s$.

(10) A message from $T'.r$ for $T'.s$ is received.

(a) Increment the message number in the *mark* field on the DOA from $M(T',r,s)$ to $T'.r$ by 1. Set the *ioas* field to null.

(b) If there is an arc from $T'.s$ to $M(T',r,s)$, delete the arc, abort the associated timeout if *timed_out* is *FALSE*, and supply the message to $T'.s$; otherwise queue the message. If there are no remaining outgoing arcs from $T'.s$ set its status to *active*.

Timeouts, Forward and Backward Propagation

(11) Timeout on an internal arc a completes.

(a) Set the *timed_out* field on the arc to *TRUE*.

(b) Check for a cycle of internal arcs involving the arc a. If one or more such cycles exist, abort the transaction involved in the arc and stop. (Aborting the transaction involves releasing the resources held by the agents of the transaction, deletion of the nodes representing

the agents, deletion of the arcs incident at these nodes and abortion of associated timeouts, as necessary).

(c) Traverse the graph forwards from the head of a, along internal arcs with their *timed_out* fields *TRUE*, to find all the DOAs and associated IOAs that can be reached. Let $ROA(a,*) = \cup[ROA(a,v1),ROA(a,v2),....ROA(a,vp)]$, be the set of arc-identifiers corresponding to these DOAs and associated IOAs, where $ROA(a,vi)$ is the set of arc-identifiers corresponding to DOAs and associated IOAs reachable by the above procedure from a whose head nodes are local to site vi. Let $ROA'(a,*)$ be computed in a similar way to $ROA(a,*)$ with the added restriction that only DOAs with *timed_out* fields set to *TRUE* and their associated IOAs are considered.

(d) Traverse the graph backwards from the tail of a, along internal arcs with their *timed_out* fields *TRUE*, to find all the DIAs with their *timed_out* fields also *TRUE* that can be thus reached. Let $RIA(a,*)=\cup[RIA(a,w1),RIA(a,w2),....RIA(a,wq)]$, be the set of these DIAs, where $RIA(a,wj)$ is the set of DIAs at site s, whose *timed_out* fields are *TRUE*, whose tail nodes are local to site wj and from the heads of each of which, a path of internal arcs with *timed_out* fields set to *TRUE*, leads to the tail of a.

(e) If one of the members of $\{v_i\}$, say v_p, is the site s itself, and $ROA(a,v_p)$ contains a member that corresponds to a DIA in $RIA(a,*)$, then a deadlock exists, hence abort the transaction involved in a and stop.

If $RIA(a,*)$ is non-empty then

(i) Backward Propagation: if $ROA'(a,*)$ is non-empty, then for

each site wj send a BP signal with IA_SET set to $RIA(a,wj)$ and OA_SET set to $ROA(a,*)$;

(ii) Forward Propagation: if $ROA(a,*)$ is non-empty, then for each site v_i, send a FP signal with ORIGIN set to s, VICTIM set to the transaction associated with the arc a, CHECK_SET set to $RIA(a,*)$, TRAVERSED_SET set to $ROA(a,*)$ and OA_SET set to $ROA(a,v_i)$.

COMMENT:Suppose a chain of arcs consisting of a DIA d_i, zero or more internal arcs and a DOA d_o, is formed at the site s. This will result in the sending of a BP signal to the site to which the tail of d_i is local. In order to prevent possible multiple identical BP signals from being sent when the timeouts on the arcs on this chain complete, the algorithm is designed so that the last of the arcs on the chain to have its *timed_out* field set *TRUE* is the only one whose timeout completion results in a BP signal being sent.

A similar situation exists in the case of the FP signals. However, the completion of the timeout on a DOA does not initiate a FP signal (it would be redundant, since there is a DIA corresponding to the DOA, at another site, whose timeout completion would trigger FP signals if necessary). Hence, here the algorithm calls for an FP signal to be sent when the last arc on the chain ***excluding*** d_o, has its *timed_out* field set *TRUE*.

(12) Timeout completes on DIA, d_i.

(a) Set *timed_out* field of d_i to *TRUE*.

(b) Let $RIA(d_i,*)=RIA(d_i,w) = d_i$ where w is the site to which the tail

of d_i is local. Compute $ROA(d_i,*)$ and $ROA'(d_i,*)$ analogously to step 11(c) above, starting the graph traversal from the head of d_i.

(c) Perform backward and forward propagation in the same manner as in step 11(e).

(13) Timeout completes on DOA, d_0.

(a) Set *timed_out* on d_0 to *TRUE*.

(b) Let $ROA'(d_0,*)$ = (arc-identifier of d_0) $\cup$ (*ioas* field of d_0). Compute $RIA(d_0,*)$ analogously to step 11(d), starting the backward traversal from the tail of d_0.

(c) If $RIA(d_0,*)$ is non-empty, perform backward propagation in the manner described in step 11(e)(i).

(14) An FP signal f is received.

(a) From OA_SET(f), check which members tally with the arc-identifier of a DIA whose *timed_out* field is set *TRUE*. Let $X=\{d_i\}$ be the set of such DIAs.

(b) For each d_i in X determine $ROA(d_i,*)$ as in step 11(c). If the head of any member of $ROA(d_i,*)$ is local to the site ORIGIN(f), and the member tallies with a member of CHECK_SET(f), a deadlock exists, hence abort transaction VICTIM(f) and stop. Delete those arc-identifiers from $ROA(d_i,*)$ that tally with a member in TRAVERSED_SET(f). Let the remaining set of arc-identifiers be designated $XROA(d_i,*)$. Let $S = \{OA(v_1),OA(v_2),.....OA(v_n)\}$ be the union of $XROA(d_i,*)$ for all d_i in X, partitioned according to the sites $\{v_j\}$ to which the heads of the outgoing arcs are local.

(c) For each v_j, send an FP signal to site v_j with ORIGIN set to

ORIGIN(f), CHECK_SET set to CHECK_SET(f), TRAVERSED_SET set to TRAVERSED_SET(f) $\cup S$ and OA_SET set to OA(v_j).

(15) A BP signal b is received.

For each member d of IA_SET(b), if a DOA d_0 exists that tallies with d,

(i) add the members of OA_SET(b) to the *ioas* field of the DOA, ignoring those whose head is the same as the tail of the DOA. (This situation can arise when a cycle exists.)

(ii) if the *timed_out* field of d_0 is *TRUE*, compute $RIA(d_0,*)$ analogously to step 11(d), performing the backwards graph traversal from the tail of d_0. Then, if $RIA(d_0,*)$ is non-empty, send BP signals as in 11(e)(i),with the OA_SET set to the set of new members in the *ioas* field of d_0.

The timeout periods of $T1, T2, T3, T4, T5$ are started when an agent is waiting to acquire a lock on a local resource, an agent is waiting to acquire a lock on a remote resource, a resource is waiting to be released by a local agent, a resource is waiting to be released by a remote agent and when an agent is waiting to receive a message. They should therefore have values appropriately in excess of the average times required for these respective events to occur. It is clear that $T2-T1$ and $T4-T3$ should be of the order of a message delay.

4.6.2.3. An Example of Deadlock Processing

In this section, we give an example to show how the algorithm works. The same conventions are used in Fig. 4.7.a-g as in Figs. 4.5 and 4.6, except that the value of the *timed_out* field on each arc is also shown (T and F stand

for *TRUE* and *FALSE* respectively). Further the flow of BP and FP signals is also shown. There are 3 sites A, B, C in this example.

Fig. 4.7.a: The transaction agents $T1.A$, $T1.B$ and $T1.C$ request locks on local resources $R1$, $R2$, $R3$ respectively and acquire them. Since no outstanding arcs from the agents remain, no timeouts are started and the *timed_out* fields are set to *TRUE* as soon as the arcs are created.

Fig. 4.7.b: The following events have occurred since the situation depicted in Fig 4.7.a existed.

(i) $T2.B$ and $T3.C$ requested locks on remote resources $R3$ and $R1$ respectively at times 60,61 respectively. This leads to these agents' status being set to *waiting* and to the creation of DOAs at sites B and C respectively. The *timed_out* field on each of these arcs is set to *FALSE* and timeouts are started on both of them. RR signals are sent to C and A respectively.

(ii) The RR signals are received. Since the requested resources are unavailable, DIAs are created at sites C and A with their *timed_out* fields set to *FALSE* and timeouts are started on the DIAs.

(iii) $T1.A$ requests creation of an agent $T1.B$ at site B. This leads to the creation of appropriate message nodes and incident arcs at site A. An AC signal is sent to site B to create an agent $T1.B4$.

(iv) On receiving the AC signal, appropriate message nodes and incident arcs are created at site B.

Fig. 4.7.c: (i) $T1.B$ has requested a lock on $R2$. Since $R2$ is unavailable, the status of $T1.B$ is set to *waiting*, the *timed_out* field on the arc from $T1.B$ to $R2$ is set to *FALSE* and a timeout started.

(ii) At site A, $T1.A$ completes its local processing and now waits to receive a

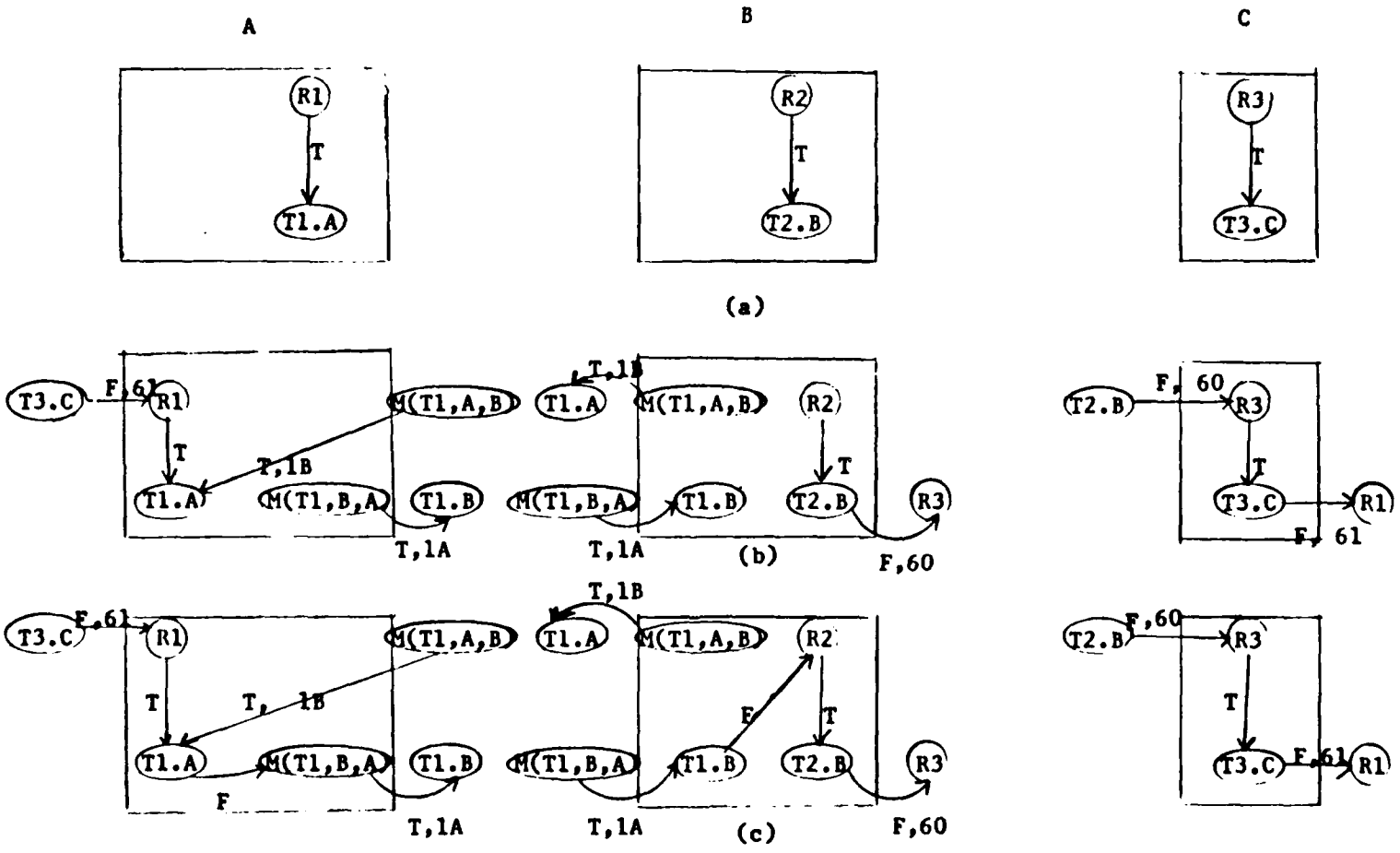


FIG. 4.7. EXAMPLE TO ILLUSTRATE THE WORKING OF THE DISTRIBUTED ALGORITHM

message from $T1.B$. It enters *waiting* status, an arc from $T1.A$ to $M(T1,B,A)$ is created, its *timed_out* field set to *FALSE* and a timeout is started.

Fig. 4.7.d:(i) The timeouts on the DIAs $T3.C \rightarrow_{61} R1$ at site A and $T2.B \rightarrow_{60} R3$ at site C complete, and therefore the *timed_out* fields are set to *TRUE*. Conditions are now satisfied for site C to send an FP signal to site A with its CHECK_SET containing the DIA $T2.B \rightarrow_{60} R3$ and its OA_SET containing the arc-identifier $(R1,61)$.

(ii) On receipt of this FP signal, site A finds a DIA with its *timed_out* field set to *TRUE* and its arc-identifier tallying with the arc-identifier in the OA_SET. However, there is no DOA reachable from this DIA through a path of internal arcs with their *timed_out* fields *TRUE*. Hence, this deadlock computation stops here.

Fig. 4.7.(e) (i) The timeouts on the DOAs $T2.B \rightarrow_{60} R3$ at site B and $T3.C \rightarrow_{61} R1$ at site C complete and therefore the *timed_out* fields are set to *TRUE*. A BP signal is sent by site C with IA_SET containing the DIA $T2.B \rightarrow_{60} R3$ and OA_SET the arc-identifier $(R1,61)$.

(ii) On receiving this BP signal, site B finds that the IA_SET member matches a DOA and hence the arc-identifier $(R1,61)$ is added to the *ioas* field of this DOA. Backward propagation from site B is inhibited since the arc from $T1.B$ to $R2$ has not completed its timeout.

Fig 4.7.f (i): The timeout period on the arc from $T1.B$ to $R2$ completes and the *timed_out* field is set to *TRUE*. The following signals get sent:

—an FP signal f1 to site C with ORIGIN set to B, VICTIM set to $T1$, TRAVERSED_SET set to $\{(R3,60),(R1,61)\}$, OA_SET set to $\{(R3,60)\}$ and

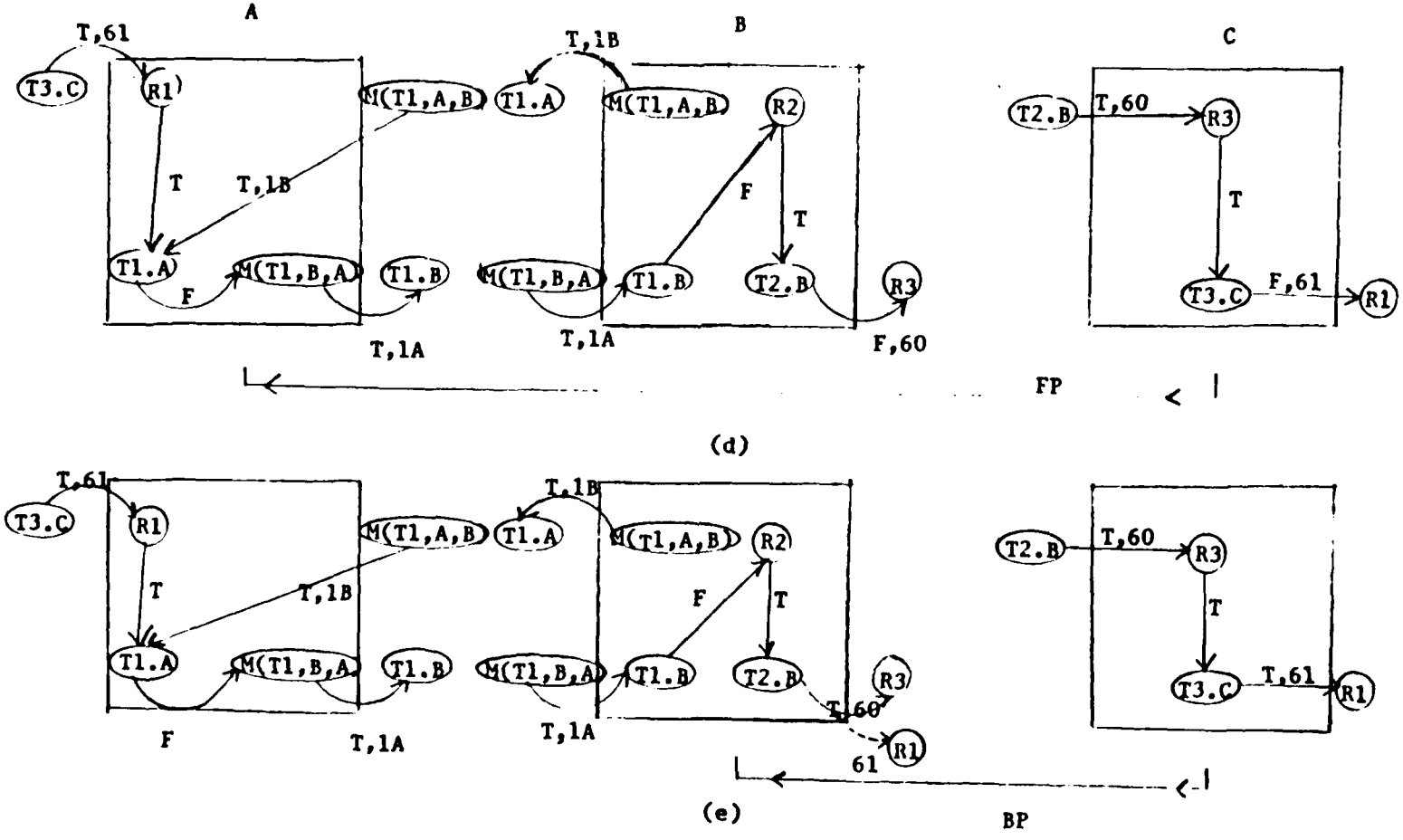


FIG 4.7. EXAMPLE TO ILLUSTRATE THE WORKING OF THE DISTRIBUTED ALGORITHM (Contd.)

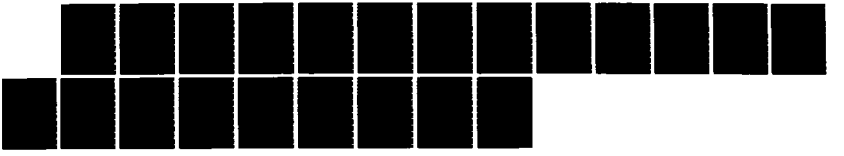
AD-A169 [redacted]7 AVAILABILITY AND CONSISTENCY OF GLOBAL INFORMATION IN COMPUTER NETWORKS(U) CALIFORNIA UNIV BERKELEY C V RAMAMOORTHY MAY 86 ARO-19159.3-EL DAAG29-83-K-0086

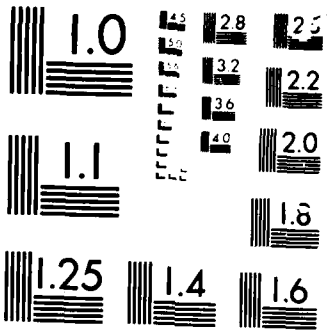
3/3

UNCLASSIFIED

F/G 9/2

NL





CHECK_SET set to $\{M(T1,B,A) \rightarrow_{1A} T1.B\}$.

–an FP signal f2 to site A the same as above, except that OA_SET is set to $\{(R1,61)\}$.

–a BP signal b1 to site A with IA_SET set to $\{M(T1,B,A) \rightarrow_{1A} T1.B\}$ and OA_SET set to $\{(R3,60),(R1,61)\}$.

(ii) The FP signal f1 reaches site C. The OA_SET member matches the sole DIA at C. However, although its sole DOA is reachable by a path of internal arcs with their *timed_out* fields set *TRUE* from this DIA, its arc-identifier is included in the TRAVERSED_SET field of f1, hence no FP signal is generated.

(iii) The FP signal f2 reaches site A. Although the OA_SET member matches a DIA, no DOA is reachable from this DIA through a path of internal arcs with their *timed_out* fields set *TRUE*, and again no FP signal is generated.

If the BP signal b1 reaches site A and is processed before the timeout period on the arc $T1.A \rightarrow M(T1,B,A)$ completes, then the deadlock will be detected locally when the latter event occurs. In our example, the BP signal does *not* reach site A in time for this to occur.

Fig. 4.7.g: (i) The timeout period on the arc from $T1.A$ to $M(T1,B,A)$ completes and the *timed_out* field is set to *TRUE*. This leads to two signals being sent:

– an FP signal $f3$ to site B with VICTIM set to $T1$, ORIGIN set to A, OA_SET and TRAVERSED_SET set to $\{(T1.B,1A)\}$ and CHECK_SET set to $\{T3.C \rightarrow_{61} R1\}$.

– a BP signal b2 to site C with OA_SET set to $\{(T1.B,1A)\}$ and IA_SET set to $\{T3.C \rightarrow_{61} R1\}$.

(ii) On receipt of f3, site B finds that the OA_SET member matches its DIA $M(T1,B,A) \rightarrow_{1A} T1.B$. From this DIA the DOA $T2.B \rightarrow_{80} R3$ and its associated

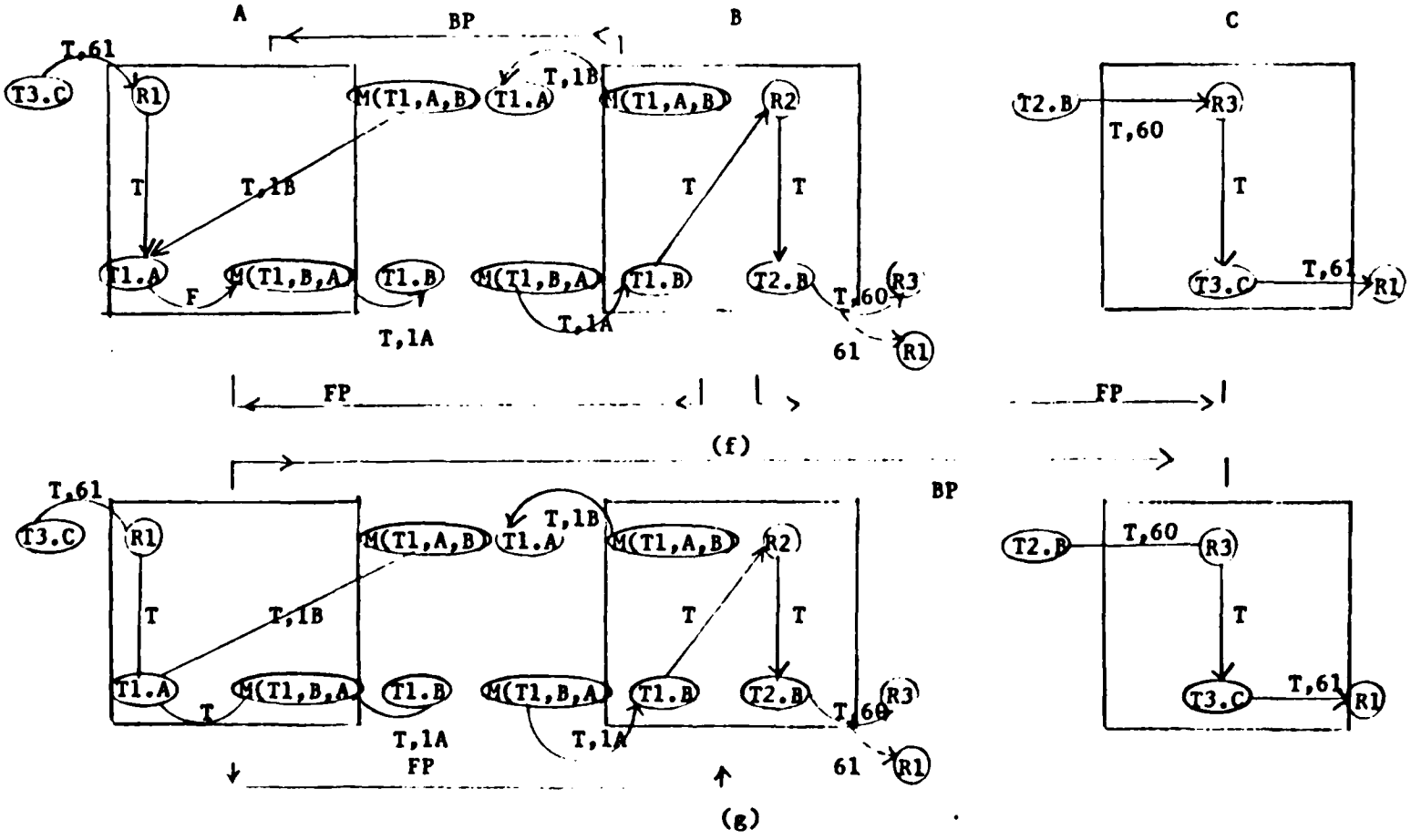


FIG. 4.7. EXAMPLE TO ILLUSTRATE THE FUNCTIONING OF THE DISTRIB TED ALGORITHM (Contd.)

IOA with arc-identifier ($R1,61$) can be reached through a path of internal arcs on which the *timed_out* fields are set to *TRUE*. But the arc-identifier ($R1,61$) tallies with the CHECK_SET member of the FP signal. Hence a deadlock is declared and transaction $T1$ aborted.

4.6.2.4. Proofs of Correctness

In this section we show that the algorithm detects all genuine deadlocks and does not give any indications of deadlock when none exists.

Thm 4.4: Every genuine multisite deadlock is detected by the above algorithm.

Proof: Suppose there is a global deadlock cycle: $e(1) \rightarrow e(2) \rightarrow e(3) \ldots \rightarrow e(m)$ where $e(.)$ is a transaction agent, a resource or message node, with each node being distinct.

Each dependency is represented as an internal arc or DIA at some site. The cycle takes the form of a chain of arcs C: $\{d_i(s_1), i(s_1,1), i(s_1,2), \ldots, i(s_1,n_{s_1})\}$ $\{d_i(s_2), i(s_2,1), i(s_2,2), \ldots, i(s_2,n_{s_2})\} \ldots\ldots \{d_i(s_m), i(s_m,1), i(s_m,2), \ldots\ldots, i(s_m,n_{s_m})\}$.

The arcs enclosed within a pair of curly braces represent a contiguous portion of the cycle contained at one site, with neighboring portions of the cycle being at a different site. Each such contiguous portion consists of a DIA and zero or more internal arcs. The head of the last arc is the tail of a DOA that coincides with the DIA for the next portion of the cycle.

Every transaction agent in C is blocked. Inspection of the algorithm shows that when an agent in C enters *waiting* state, and the arc in C that is directed away from the agent is created a timeout period is started for the arc. Hence at least one timeout period is started in connection with the

creation of an arc in C. Since C represents a deadlock, a timeout started will not be aborted and will complete

Let a be the arc in C whose timeout is last to complete among all the timeouts that are started in connection with the creation of arcs in C and let t be the time this last timeout completes. We claim that all arcs in C will have their *timed_out* fields set *TRUE* by t. By definition of t, all arcs in C that have timeouts started on their creation, will have their timeouts complete and their *timed_out* fields set *TRUE* by t. But each arc in C that does not have a timeout started on its creation, occurring at t1, and therefore has its *timed_out* field set *TRUE* immediately at t1, must, by inspection of the algorithm, be an internal arc indicating resource possession or a DIA indicating message possession. The transaction agent at the head of this arc must at a time equal to or greater than t1, enter *waiting* state as a result of requesting the resource or message to which there is an arc in C from the agent. A timeout will be started for this arc (by inspection of the algorithm) which completes at $t2 < t$. Since $t1 \leq t2$, $t1 < t$.

Let the arc a be in site s_1 (without loss of generality). At t, an FP signal f1 will be sent to s_2 with the arc-identifier of $d_i(s_2)$ in its OA_SET and $d_i(s_1)$ in its CHECK_SET (unless the arc-identifier of $d_i(s_1)$ is present as an IOA reachable from a, in which case the deadlock is detected locally). If on receiving this signal, s_2 does not send an FP signal f2 to s_3 with the arc-identifier of $d_i(s_3)$ in its OA_SET and $d_i(s_1)$ in its CHECK_SET, it will be either because
(i) the TRAVERSED_SET of f1 already contains this arc-identifier, which means that s_1 simultaneously with sending f1 to s_2 sent an FP signal f1' to s_3 with the arc-identifier of $d_i(s_3)$ in its OA_SET, or because
(ii) the deadlock cycle is detected at s_2, as a result of the presence, as an IOA

reachable from $d_i(s_2)$, of the arc-identifier of $d_i(s_1)$.

In any case, unless the deadlock is detected at s_1 or s_2, site s_3 will receive an FP signal with the arc-identifier of $d_i(s_3)$ in its OA_SET and $d_i(s_1)$ in its CHECK_SET. Proceeding in this manner, we can show that unless the deadlock is detected at one of the sites $s_1,...,s_{m-1}$, an FP signal will be received by s_m with $d_i(s_m)$ in its OA_SET and $d_i(s_1)$ in its CHECK_SET. But the existence of the cycle C means that there is a DOA at s_m reachable through a path of internal arcs with their *timed_out* fields set *TRUE* and corresponding to $d_i(s_1)$. Hence the deadlock will be detected.

–

Thm 4.5: No false indications of multisite deadlock are given by the given algorithm.

Proof: Suppose a multisite deadlock is detected at the origin of a deadlock computation, as a result of the presence of an IOA reachable from a DIA, which corresponds to the IOA itself. The presence of such an IOA means that the DIA cannot vanish until after it vanishes, i.e. it will not vanish (except by transaction aborts). Hence the deadlock is genuine.

Suppose that a deadlock be detected as the result of the following sequence of FP signals f0,f1,f2....,f(l-1), originating at sites s_0, s_1, s_2,.....,s_{l-1} and sent to sites s_1, s_2, s_3,....s_l respectively. Receipt of fj (j=0,1,2...,l-2) triggers sending of f(j+1). Let the sending of fj (j=0,1,....,l-1) occur at time tj. Consider the set of DIAs in the CHECK_SET of f0. At t0, s_0 has not sent the signals that will accompany the deletion of any of these DIAs. Further, it can do this only after all the arcs corresponding to the arc-identifiers in the OA_SET of f0 have been deleted at s_0.

At t_1, when f0 is received at site s_1, it is found that one or more entries in OA_SET(f0) coincide with DIAs in s_1. Unless and until these DIAs are deleted at s_1 and the corresponding signals sent to s_0, it is not possible to delete all the DOAs in OA_SET(f0). But all these DIAs cannot be deleted before all the arcs corresponding to the arc-identifiers in OA_SET(f1) are deleted. Thus it is not possible to delete any of the DIAs in CHECK_SET(f0) before all the arcs corresponding to the arc-identifiers in OA_SET(f1) are deleted.

Proceeding in this manner, we conclude that none of the DIAs in CHECK_SET(f0) can be deleted before all the arcs corresponding to the arc-identifiers in OA_SET(f(l-1)) are deleted.

Since the deadlock is detected at site s_m on receipt of f(l-1)) it follows that:

(i) at the time of declaration of deadlock, there are DIAs in s_i that coincide with *members of* OA_SET(f(l-1)).

(ii) from these coincident DIAs, it is possible to reach, at the time of declaration of deadlock, DOAs or IOAs that are coincident with members of CHECK_SET(f0).

Hence, using the same reasoning as above, it follows that these coincident members of CHECK_SET(f0) cannot be deleted until after they are deleted, which implies that they cannot be deleted at all (unless by transaction aborts). Hence the deadlock indication is correct.

–

4.6.2.5. Performance

The proposed algorithm detects multisite deadlocks faster than other algorithms that have been proposed in the literature. This is because both forward and backward propagation are used on this algorithm.

In the worst case, when all the wait dependencies in an intersite cycle form at approximately the same time, the time required to detect the deadlock is approximately half the time it would require to go round the cycle. All distributed schemes proposed so far require a detection time equal to, if not greater than, the time to go round the cycle.

If the last wait dependency occurs after a substantially long time from the rest of the wait dependencies in the cycle, the deadlock will be detected locally without having to go round the cycle. For cases in between the two extreme cases cited above, the detection time will be intermediate.

The penalty paid for the improvement in response time is higher message traffic. For a deadlock involving n sites, our algorithm requires a maximum of approximately n^2 FP signals. (Each site may send one signal to each of the other n-1 sites, serially if no IOAs are present (the signals flowing around the loop) or in a combination of sequential flow and parallel flow if IOAs are present). Algorithms proposed till now use only serial flow and in such algorithms it is possible to reduce the amount of communication by half by requiring a serial flow of messages originating from a given transaction to stop when it encounters a transaction of higher id than the originating transaction. This optimization is difficult to incorporate in our algorithm, in which FP signals can "skip over" one or more sites, and thus over the nodes in the cycle in those sites.

Secondly, there is the overhead of backwards propagation which is also approximately of the order of n^2 signals for a cycle encompassing n sites.

In practice, chains of dependencies spreading over a large number of sites are unlikely to occur except under conditions of severe contention. It appears reasonable to pay the cost of higher communication traffic at such times in order to quickly detect any deadlock that may exist and which if not detected for a longer time, would exacerbate the contention.

4.7. Conclusion

In this chapter, we have discussed centralized and distributed detection of deadlocks in a distributed system. For the case of centralized detection, we showed that reports only from resource-controlling sites are sufficient, if usage of resources is 2-phase, to detect deadlocks correctly. For the case where detection is centralized but resource usage is not 2-phase, we constructed an algorithm for the "migrating" transaction model, in which non-resource-controlling sites are only rarely required to participate in deadlock detection. Since resource-controlling sites will be generally few compared to the number of sites accessing the resources, the communication costs for deadlock detection are sharply reduced by this algorithm. Lastly, we constructed a distributed detection algorithm which uses both forward and backward traversal of the transaction_agent-resource-message graph to speed up deadlock detection.

The algorithms utilize clock facilities to address the problem of race conditions. In the centralized algorithm, timestamps derived from the clock facility are assigned to every request for a resource. When the deadlock detector site assembles reports from the resource controllers in the system, it uses these timestamps to ascertain if an observed cyclic wait represents a

genuine deadlock. In the distributed scheme, timestamps (or message numbers concatenated with site id) are affixed to every intersite arc. This allows use of "condensed" information to hasten the detection of deadlocks. Without the timestamps, it would be difficult to ascertain if the condensed information is up-to-date. It is difficult, as pointed out in [GLI 80] to update this condensed information soon enough to prevent spurious indications of deadlock. By affixing marks as mentioned above to the condensed information, the urgency of updating it is removed.

CHAPTER 5

CONCLUSION

In this report, we addressed the problems of maintaining the availability and consistency of global information in computer networks. Below, we summarize our results, describe some experiences during the research and suggest future directions.

In Chapter 2, we described a network status maintenance scheme based on a global clock mechanism. Our scheme differs from that of [HAM 80] in that it relies upon the nearest neighbors of a site to determine its status and propagate it, whereas in the latter scheme, probe messages are sent by any site to determine the status of another site. An important lesson we learned was how to put together reports from the neighbors of a site to determine its status. The problem here is that all the links to a site may appear dead at different times to its neighbors, but the site itself may never have crashed or noticed that it was partitioned from the rest of the network. But these status reports may be put together at another site which may then conclude that the site has crashed. Rule *C*3 would then be violated. The natural approach to solving this problem seemed to be to require that the report timestamps should lie within a small time window. However, difficulties were encountered in ensuring that a partitioned site observed its links to the rest of the network to be down in a similar time window and crashed itself in time to comply with rule *C*3. The solution adopted in the end involved putting together the *latest* reports from all neighbors of a site to determine if the site should be marked *DOWN*. The problem mentioned above was solved by

having a site mark a recovered link to a neighbor as *UP* in its *CRASH_SELF* graph only after all other sites have marked it *UP* in their *CRASH_OTHERS* graphs.

A promising extension of our approach is in the direction of dynamic networks. Such networks will typically consist of non-overlapping clusters of sites, each cluster functioning more or less independently of other clusters. Sites may migrate from one cluster to another. In order to maintain the consistency of information concerning membership of sites in clusters, the following modification of rule *C*3 seems appropriate:

*C*3': If site x in cluster *C* does not have site y included in its list of sites in *C* at time t, then site y does not consider itself to be a member of *C* at time t.

Our approach to realizing rule *C*3 in static networks, described in Chapter 2, suggests how *C*3' may be realized in a dynamic network. Site x would remove y from its list of sites in *C* only when it finds that sites in *C* are unable to communicate with y. Site y finding that it has lost contact with sites in *C* would consider its membership in *C* to have lapsed. It would cease to carry out the actions it was carrying out as a member of *C*, and institute appropriate recovery actions, e.g., reapplying for membership in *C*, becoming a member of another cluster, etc.

In searching for a control problem to test out our network status maintenance scheme, we found that many problems become simple to solve using the scheme. An example is the *election* protocol of [GAR 82]. The problem here is to choose a unique co-ordinator for a group of sites, when the current co-ordinator crashes. At all times it is desired to have the site with highest id which is *UP* as co-ordinator. The solution given is to make the election of a co-ordinator *atomic* by using a 2-phase protocol to broad-

cast the id of the new co-ordinator, similar to the 2-phase protocol in distributed database systems [GRA 78]. The proof that the protocol works correctly is not simple. But using our network status maintenance scheme, the solution is quite straightforward: a site simply considers the site with highest id marked *UP* in its *CRASH_OTHERS* graph as the current co-ordinator. Rule *C*3 then ensures that no two sites consider themselves co-ordinator at the same time.

We developed a solution to the replicated file update problem in Chapter 2. Without the use of the clock facility, it would be difficult to ensure that two (or more) different *WARM* sites do not join the set of *HOT* sites when a *HOT* site crashes. If two different *WARM* sites do join, it would be necessary either to force one and exactly one of them to quit the set subsequently, or else to incur the overhead of updating an extra site before a 'done' signal is returned to the originator of an update transaction.

In Chapter 3, we addressed the problem of preventing error propagation in global information due to malfunctioning sites. We found that a more general form of the Byzantine Generals Agreement was required and formulated it. The notion of different kinds of *malfunction–tolerance–specification* was introduced as a way to trade off tolerance to malfunctions against the costs involved. There are still many areas where knowledge of ways to provide robustness against malfunctions is inadequate. These include synchronization, security, efficient transfer of bulk data, update interactions involving more than one updated variable, etc. A prototype for testing out BGA algorithms is currently under construction at the San Jose IBM Research Laboratory [STR 82] and experience in this project should contribute in this direction.

In Chapter 4, we addressed deadlock detection in distributed databases. Algorithms for centralized and distributed detection were proposed. For centralized detection it was shown that 2-phase usage of resources simplifies the problem of dealing with race conditions. An efficient centralized algorithm was proposed for the 'migrating' transaction model. A distributed algorithm using both backward and forward propagation to hasten detection was described. In both algorithms, a clock facility was the means whereby the consistency of 'snapshots' of system status was ensured.

REFERENCES

[AST 76] Astrahan, M.M. et al, "System R: Relational Approach to Database Management", *ACM Transactions on Database Systems*, Vol. 1, No. 2, 1976.

[BAD 83] Badal, D.Z., and Gehl,M.T., "On Deadlock Detection in Distributed Computing Systems", *Proc. COMPSAC*, pp 36-45, 1983.

[BAR 65] Barlow, R.E., and Proschan, F., "Mathematical Theory of Reliability", Wiley, New York, 1965.

[BEL 79] Belford, G.G. and Grapa, E. "Setting Clocks 'back' in a Distributed Computing System," *1st Intl. Conf. Distributed Computing Systems*, Huntsville, Ala., Oct. 1979.

[BER 80] Bernstein, P.A., Shipman, D.W., and Rothnie,J.B., "Concurrency Control in a System for Distributed Databases (SDD-1)", *ACM Transactions on Database Systems*, pp 18-25, March 1980.

[BER 81] Bernstein, P.A. and Goodman,N., "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys*, pp 185-222, June 1981.

[CAR 83] Carey, M.J., "Modeling and Evaluation of Database Concurrency Control Algorithms", Ph.D. thesis, University of California at Berkeley, Sept. 1983.

[CHA 74] Chandra, A.N. et al, "Communication Protocols for Deadlock Detection in Computer Networks", IBM Technical Disclosure Bulletin, vol. 16, no. 10, March 1974.

[CHA 82] Chandy, K.M., and Misra,J., "Deadlock Detection in Distributed Databases", Tech. Report TR-LCS-8205, Department of Computer

Sciences, The University of Texas at Austin, March 1982.

[DIF 76] Diffie, W. and Hellman, M., "New Directions in Cryptography", *IEEE Transactions on Information*, June 1976.

[DOL 81] Dolev, D., "Unanimity in an Unknown and Unreliable Environment," *22nd Annual Symposium on Foundations of Computer Science*, 1981.

[DOL 82a] Dolev, D., "The Byzantine Generals Strike Again," *Journal of Algorithms*, vol. 3, no. 1, 1982.

[DOL 82b] Dolev, D., Fischer, M., Fowler, R., Lynch, N. and Strong, H.R., "Efficient Byzantine Agreement without Authentication," submitted for publication.

[DOL 82c] Dolev, D. and Strong, H.R., "Authenticated Algorithms for Byzantine Agreement," IBM Research Report RJ3416 (1982).

[ESW 76] Eswaran, K.P. et al, "The Notions of Consistency and Predicate Locks in a Database System", *Communications of the ACM*, November 1976.

[GAL 82] Galler, B., "Concurrency Control Performance Issues", Ph.D. thesis, University of Toronto, Sept. 1982.

[GAR 82] Garcia-Molina, H., "Elections in a Distributed Computing System", *IEEE Trans. on Computers*, pp 48-59, January 1982.

[GIF 79] Gifford, D.K., "Violet: An Experimental Decentralized System," *Operating Systems Review*, vol. 13, no. 5, 1979.

[GLI 80] Gligor, V. and Shattuck,S., "On Deadlock Detection in Distributed Systems", *IEEE Transactions on Software Engg.*, pp 335-340, Sept. 1980.

[GOL 77] Goldman, B., "Deadlock Detection in Computer Networks", Technical Report, M.I.T./LCS TR-185, Sept. 1977.

[GRA 78] Gray, J.N., Notes on Database Operating Systems, in *Operating Systems: An Advanced Course*, Lecture Notes in Computer Science 60, Springer Verlag, 1978, pp 393-481.

[GRA 81] Gray, J., "The Transaction: Virtues and Limitations", Tandem Tech. Rep. TR 81.3, June 1981.

[HAB 69] Habermann, A.N., "Prevention of System Deadlocks", *Communications of the ACM*, pp 373-377, July 1976.

[HAM 80] Hammer, M. and Shipman, D., "Reliability Mechanisms for SDD-1: A System for Distributed Databases," *ACM Transactions on Databases*, Dec. 1980.

[HO 79] Ho, G.S.,"A Systematic Approach for the Design and Analysis of Distributed Computer Systems," Ph. D. Dissertation, University of California at Berkeley, 1979.

[HOL 72] Holt, R.C., "Some Deadlock Properties of Computer Systems", *Computing Surveys*, pp 179-196, Sept 1972.

[HON 79] Reference Manual for the ADA Programming Language, Honeywell Inc., Minneapolis and Cii Honeywell Bull, Louviciennes, France, March 1979.

[ISL 78] Isloor, S.S., and Marsland, T.A., "An Effective "On-Line" Deadlock Detection Technique for Distributed Data Base Management Systems", *Proc. COMPSAC* 1978, pp 283-288.

[ISL 80] Isloor, S.S., and Marsland, T.A., "The Deadlock Problem: An Overview", *Computer*, pp 58-77, Sept. 1980.

[KUH 80] Kuhl, J.G., Reddy, S.M., "Distributed Fault-Tolerance For Large Multiprocessor Systems", *Proc. of 7th Symp. on Computer Architecture*, May 1980.

[LAM 76] Lampson, B. and Sturgis, H.E., "Crash Recovery in a Distributed Data Storage System", Internal Report, Comput. Sci. Lab, Xerox Palo Alto Research Center, Palo Alto, Ca, 1976.

[LAM 78a] Lamport, L., "Time, Clocks and the Ordering of Events in a Distributed System", *Communications of the ACM*, vol. 21, pp 558-564, July 1978.

[LAM 78b] Lamport, L., "The Implementation of Reliable Distributed Multiprocess Systems", *Computer Networks*, vol 2, May 1978.

[LAM 80] Lamport, L., Shostak, R. and Pease, M., "The Byzantine Generals Problem," Technical Report, Computer Science Lab., SRI Intl.,1980.

[LAM 81a] Lamport, L., "Using Time instead of Time-out for Fault-Tolerant Distributed Systems," Technical Report, Computer Science Lab., SRI Intl., 1981.

[LAM 81b] Lamport, L., and Melliar-Smith, P.M., "Synchronizing Clocks in the Presence of Faults", Tech Report, Computer Science Lab., SRI Intl., 1981.

[LAM 81c] Lampson, B.W., Applications and Protocols, in *Distributed Systems: Architecture and Implementations, An Advanced Course*, Chapter 14, Springer Verlag, 1981.

[LIN 83] Lin, W. and Nolte, J., "Basic Timestamp, Multiple-Version Timestamp, and Two-phase Locking", *Proceedings of Symposium on*

Reliability in Distributed Software and Database Systems, Palo Alto, Ca, Oct. 1983.

[MA 81] Ma, Y., "Techniques for the Design and Management of Dynamic Computer Networks", Ph.D. Dissertation, Univ. of California, Berkeley, 1981.

[MAH 76] Mahmoud,S. and Riordon,J.S., "Protocol Considerations for Software Controlled Access Methods in Distributed Databases", *Proceedings of the International Symposium on Computer Performance Modeling, Measurement and Evaluation*, Harvard University, Cambridge, March 1976.

[MCQ 80] McQillan,J.M., Richer,I., and Rosen,E.C., "The New Routing Algorithm for the ARPANET", *IEEE Trans. on Comm.*, May 1980.

[MEN 79] Menasce,D. and Muntz,R., "Locking and Deadlock Detection in Distributed Databases", *IEEE Transactions on Software Engineering*, May 1979, pp 195-202.

[NEL 81] Nelson, B.J., "Remote Procedure Call", Ph.D. Thesis, Carnegie-Mellon University, May 1981.

[OBE 82] Obermarck, R., "A Distributed Deadlock Detection Algorithm", *ACM Transactions on Database Systems*, pp 187-209, June 1982.

[OUS 80] Ousterhout, J.K., "Partitioning and Cooperation in a Distributed Multiprocessor Operating System: Medusa", pp. 18-20, Ph.D. Dissertation, Carnegie-Mellon University, 1980.

[PEA 80] Pease, M., Shostak, R. and Lamport, L., "Reaching Agreement on the Presence of Faults," *Journal of the ACM*, vol. 27, no. 2, 1980.

[RIV 78] Rivest, R.L., Shamir, A. and Adleman, L., "A Method for obtaining Digital Signatures and Public-Key Cryptosystems", *Communications of the ACM*, January 1978.

[ROS 77] Rosenkrantz, D.J. et al, "A System Level Concurrency Control for Distributed Database Systems", *Proc. 2nd Berkeley Workshop on Distributed Database and Computer Networks*, Berkeley, Ca, May 1977.

[SCH 83] Schlichting, R.D. and Schneider, F.B., "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems", *ACM Transactions on Computer Systems*, pp 222-238, August 1983.

[SKE 81] Skeen, D., "Non-blocking Commit Protocols", *SIGMOD International Conf on Management of Data*, 1981.

[STO 79] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies in Distributed INGRES", *IEEE Transactions on Software Engg.*, May 1979.

[STR 82] Strong, H.R., and Dolev, D., "Byzantine Agreement", Technical Report RJ 3714, IBM Research Division, Dec. 1982.

[SUN 78] Sun, H.C., "Deadlock Detection in Distributed Systems- A Preliminary Evaluation", Master's Report, Dep. Comput. Sci., University of Maryland, 1978.

[THO 76] Thomas, R.H., "A Solution to the Update Problem for Multiple Copy Data Bases which uses Distributed Control", BBN Report 3340, July 1976.

[TSA 82] Tsai, W.-C. and Belford, G., "Detecting Deadlock in a Distributed System", *Proceedings INFOCOM*, pp 89-95, Las Vegas, April 1982.

[WEN 78] Wensley, J.H. et al, "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", *Proceedings of the IEEE*, vol. 60, no. 10, Oct 1978.

END

DTIC

7-86